On the Development of Next Generation Memory Exploits

Jordy Gennissen

Submitted in fulfillment for the degree of
Doctor of Philosophy

S3Lab, Information Security Group
Royal Holloway, University of London

Declaration of Authorship

I, Jordy Gennissen, hereby declare that this thesis and the work presented in it is entirely my own. Where I have consulted the work of others, this is always clearly stated.

Jordy Gennissen,
February 10, 2023

Acknowledgements

First and foremost, I would like to thank my supervisors Daniel O'Keeffe and Jorge Blasco. I am thankful for the guidance, and the freedom provided to me to research different and non-conformal topics. You stuck with supervising me during the most heated of times, and this dissertation would not have been possible without you. I would also like to thank my past supervisors, in chronological order: Lorenzo Cavallaro, Johannes Kinder and Daniele Sgandurra. Even during a limited time of supervision, I learnt a lot and I appreciated your guidance.

I am grateful for the Center for Doctoral Training at Royal Holloway, the scholarship from L3Harris TRL (formerly L3-TRL) and Royal Holloway (EP/P009301/1) that provided financial support for the past years and the hours of discussion with employees of TRL, notably Tim, Lauren, Paul and Martin.

Besides official supervisors, I have had the pleasure of collaborating with a large group of incredibly smart people throughout the world. Jassim Happa has provided great support on both a personal level and scientifically, it has been a pleasure working with you. I would like to thank Erik Poll, Veelasha Moonsamy and Konstantinos Mersinas for the long brainstorms, discussions and general guidance.

I want to thank all my colleagues at Bloomberg L.P. during my internship in 2019. I am very grateful for this opportunity provided by Julien, the fruitful work while working with Julio and Pete, and of course the great time with them, Zhenghao, Shane, and everyone else I met during this time. The internship was a once in a lifetime experience I will never forget.

Furthermore, I want to express my gratitude towards the friends and colleagues I made at Royal Holloway, in no particular order: Marcos, Nathan, Claudio, Jason, Angela, James, Amy and Mateo. Interactions may have been limited in the past years due to the pandemic and distance, but I really enjoyed the long nights, extensive conversations and all events we attended.

On a personal note, a very special thank you goes to Anna. Your love and support has helped me through some of the hardest times and brought me to the end of this journey. You understand the struggles of a PhD and you knew what I needed, even when I did not. Thank you and I love you.

I want to show upmost gratitude towards my parents — Luc & Thea — and my brothers, Milan & Sylva. Even while abroad, we have always been in touch and I believe you experienced parts of the life of a PhD student, without doing a PhD yourself. Your support has been invaluable throughout all these years and I cannot begin to imagine how I would have finished without it. Finally, I would like to say thank you to Sharon, Floor Robin and Dajo, who helped me through the hardest of times and got me back to continue the journey, to Manouk for designing and testing the Hack the Heap website, and to all people testing the website and providing feedback.

Abstract

Memory vulnerabilities can be dangerous. To counter their effects, software and hardware support is being developed for Control-Flow Integrity (CFI): a technique to stop classical exploits from working. Unfortunately, there remains an under-explored residual attack surface. We refer to exploits targeting this attack surface as *Next Generation Memory Exploits (NGMEs)*.

This dissertation focuses on attacks and defences of NGMEs through an exploitation model of three phases: vulnerability, control and payload. We discuss vulnerabilities as a whole and propose new, measurable properties for vulnerabilities. These properties form the foundation of a vulnerability taxonomy called `GEN`, as it `GEN`erically describes and classifies vulnerabilities. `GEN` defines what a vulnerability is through a technical definition. Moreover, `GEN` identifies what vulnerabilities are relevant for NGMEs and how to find them.

Typically, memory vulnerabilities expose limited capabilities. Control techniques are necessary to exploit such vulnerabilities. Heap Layout Manipulation is one technique to overwrite useful data structures. We propose a toolchain that can generate puzzles from real-world applications, to then be solved in a puzzle game called Hack the Heap. This way, we can solve the Heap Layout Manipulation problem through gamification while remaining heap manager agnostic and explainable.

With control gained over a vulnerable application, the next step is to deliver the payload: what the exploit wants to achieve. To halt this payload, we propose System Call Argument Integrity: automatic data-flow protection tailored towards security-sensitive system calls. It protects against data-only attacks while incurring overhead only when handling security-sensitive data-flows.

Concluding, we propose a novel framework for characterising NGMEs (and vulnerabilities more broadly), in addition to techniques for assessing and mitigating their impact during different phases of the exploitation lifecycle. Considering the increasing risk of real-world NGMEs, we hope this dissertation fosters further research into both NGME threats and mitigations.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ACE	Arbitrary Code Execution	47
AEG	Automatic Exploit Generation	35
AFL	American Fuzzy Lop (a fuzzer)	87
ARP	Arbitrary Read Primitive	51
ASLR	Address Space Layout Randomisation	32
ATT&CK	Adversarial Tactics, Techniques and Common Knowledge	14
AWP	Arbitrary Write Primitive	47
BOP	Block-Oriented Programming	36
CAPEC	The Common Attack Pattern Enumeration and Classification	13
CFB	Control-Flow Bending	35
CFI	Control-Flow Integrity	34
CGI	Common Gateway Interface	54
CIA	Confidentiality, Integrity and Availability	15
CTF	Capture-The-Flag	106
CVE	Common Vulnerability Enumeration	9
CVSS	Common Vulnerability Scoring System	15
CWE	Common Weakness Enumeration	14
CWP	Contiguous Write Primitive	48
DARPA	Defense Advanced Research Projects Agency	35
DEP	Data Execution Prevention	31
DFI	Data-Flow Integrity	33
DOP	Data-Oriented Programming	36
DoS	Denial-of-Service	11
DLMalloc	Doug Lea’s malloc implementation	29
GEN	The GENeric vulnerability taxonomy	59
GOT	Global Offset Table	49

HLM	Heap Layout Manipulation	25
HTH	the Hack The Heap puzzle game	97
HTTP	HyperText Transfer Protocol	43
IH	Information Hiding	21
ISA	Instruction Set Architecture	80
IU	Incorrectness-Undefinedness	64
JEMalloc	Jason Evans' malloc implementation	100
KLoC	Kilo-Lines of Code	106
LFH	Windows' Low Fragmentation Heap	25
LoC	Lines of Code	108
LTO	Link-Time Optimisation	126
ML	Machine Learning	61
MITRE	Name of the organisation, not an acronym	13
MPK	Intel's Memory Protection Keys	22
NGINX	Engine X (webserver)	106
NGME	Next Generation Memory Exploit	3
NIST	National Institute of Standards and Technology	9
NJS	NginX JavaScript	108
NX bit	Non-eXecute bit	31
OOM	Out Of Memory error	131
OS	Operating System	18
PA	The Protection Analysis project	10
PEP	Privilege Escalation Primitive	115
PTMalloc2	PThreaded adaptation of DLMalloc	29
RAM	Random Access Memory	18
RFC	Request For Comments document	60
ROP	Return-Oriented Programming	33
SCAI	System Call Argument Integrity	119
SDLC	The Software Development Life-Cycle	11
SFI	Software-Fault Isolation	21
SOC	Security Operations Center	82
SoK	Systematisation of Knowledge (body of work)	141
SQL	Structured Query Language	16

TC	Turing-Completeness	50
TCMalloc	Thread-Caching Malloc adaptation of DLMalloc	95
UaF	Use-after-Free	20
WWW/W3	Write-What-Where, see AWP	47
WX	Write-or-eXecute	31

Introduction 1

Technology has enhanced daily life in the past few decades to an extreme extent. As of 2021, an estimated 3.8 billion people own at least one smartphone [192]. Furthermore, the last years have seen incredible progress within self-driving cars [48]; over 320 million smart speakers have been installed across the world [177]; and the recent COVID-19 pandemic forced companies worldwide to change the way they work, resulting in quadrupling the amount of people working from home in the US [22] while the e-commerce business has seen a growth of 9x the average pre-pandemic growth [146].

As society becomes increasingly dependent upon digital devices, it is vital for these devices to work as intended. When a device malfunctions — whether accidental or through malice — the results can be disastrous. The compromise of smartphones and smart speakers can pose large threats to a person's privacy [17, 103, 77]; and malfunctioning self-driving cars can lead to death in extreme cases [73]. Furthermore, device malfunctions have become a vital aspect of warfare, where e.g., supply chain attacks can leave entire areas without critical resources as seen in the US in 2021 [112], while other attacks can disrupt business and communication by wiping all files from infected machines, as seen during the recent war in Ukraine [128].

Such attacks are possible when a mistake is made in developing the software. Software development is error-prone, but software development is arguably not treated as such. When we compare it against architecture, the risk of a mistake made by the architect can lead to buildings collapsing, lives being lost and millions of dollars in damage. The risk of a bug in software on the contrary is more complex, and the risks are abstract. In combination with a high demand on functionality, the process leaves room for plenty of mistakes. With most machines running on millions of lines of written software, these mistakes (i.e., bugs) are retroactively discovered on a regular basis.

Not all bugs can be used to perform damaging attacks. Thus every bug found raises the question of how problematic it is in practice. If it can be used by an attacker to compromise the software or the underlying system, it is important to patch the bug and update its software as soon as possible. Such bugs are typically referred to as *vulnerabilities* and are (or should be) prioritised over their seemingly harmless counterpart. Determining whether a vulnerability can be used by attackers is a hard task: it is generally only feasible with certainty by writing the exploit. Alternative methods have been developed to estimate the likelihood of exploitability — e.g., with CVSS — but hold no guarantees.

In today's infrastructure, correcting a single vulnerability is only *part* of mitigating the threat. Software can be deployed numerous times world-wide and is under constant development. Even if a new vulnerability is fixed, companies may still run older versions of the application, e.g., if newer versions work differently or are costly to update. In other words, vulnerabilities can pose a threat long after a fix is publicly available [113].

Next Generation Memory Exploits. In this dissertation we focus on memory-based vulnerabilities within an application. Memory-based vulnerabilities break the internal contract on where or when a piece of information ends, and where or when the next piece of information starts. A skilled exploit writer may be able to use such scenarios to read or write into other important structures, therefore compromising the application further. These vulnerabilities are known to be dangerous at least since 1972 [6] but are still regularly reported [132].

A lot has changed for memory-based exploit writers since 1972. A multitude of defence techniques are deployed today, such as Address Space Layout Randomisation (ASLR), Write-or-eXecute (WX) and stack canaries. All of the above need to be broken or circumvented if the attacker wants any chance at writing a successful exploit. Furthermore, the 64-bit architecture in modern devices is more likely to use registers over memory when possible (for example within calling conventions [181]), making memory-based vulnerabilities less likely to be effective. 64-bit pointer values may also require the exploit writer to write null-bytes, which can be notoriously difficult or even impossible.

The next defence to be deployed on a large scale is Control-Flow Integrity (CFI) [119, 120]. This will change the playing field for exploit writers once again, as current techniques seen in the wild heavily rely on the capability of hijacking the control-flow of the application.

Ahead of this development, two attack techniques have arisen in academic research. Control-flow Bending (CFB) [25, 79] breaks CFI by abusing the inherent inaccuracies of the CFI mechanism. Data-only attacks [32, 98, 156] on the other hand circumvent the CFI defence by not targeting any control data. Either technique relies upon the exact code and internal logic from the application, so they cannot be generically applied or transferred. Hereafter, we refer to a memory exploit in the presence of CFI as a "Next Generation Memory Exploit" (NGME), regardless of whether it evades or breaks CFI.

Writing NGMEs is a complex task consisting of many smaller yet still complex steps. Already, real-world exploit writing takes expertise and it can easily take weeks or months to develop one. The counterweight here is the pay-off: real-world exploits in today's digital world have much larger impact on society as a whole compared to 20 years ago. This dissertation looks at memory exploit writing with a focus on NGMEs. Specifically: how it works, how to simplify exploit writing, and how to systemically protect against these new attack techniques.

1.1 Challenges & Research Questions

Exploit writing is a difficult task done by experts. It is generally very specialised and practical in nature, and so is the communication across practitioners. For outsiders, this can be a menacing field due to its technical difficulties and limited resources. This gave rise to our main research question:

> **RQ1**: How does the exploitation process work, and how does this apply to NGMEs?

In this dissertation, the exploitation process is broken down into three phases. First, it requires a vulnerability to be exposed and known by the attacker. We then need to shape the vulnerability and use it to gain control over the application. Finally, we use the control gained to accomplish our goal: deliver the payload. Within each of the phases there are challenges we researched.

1.1.1 Challenges in the Vulnerability Phase

In an ideal world, we could eliminate all vulnerabilities in software. Unfortunately, no such thing is possible. To understand why, we need to understand what a vulnerability truly represents, but understanding the underlying cause of a vulnerability is a hard problem. If someone made a mistake (i.e., creating or exposing a vulnerability), where in the software development process was this mistake made? What type of mistake was made? Many would argue they understand vulnerabilities as a concept, despite being unable to structurally explain them beyond "a mistake was made".

Labelling vulnerabilities by the direct cause of the vulnerability can be done (e.g., "buffer overflow") but can become ambiguous (e.g., "heap overflow" or "out-of-bounds write"). For each vulnerability with a subtle difference, it requires a new class (e.g., "buffer underflow") leading to an abundance of different classes. To compensate, it can include a catch-all class (e.g., "Improper input validation") to avoid misclassifying, but this provides a large amount of interpretive space for the vulnerability to be classified. It also becomes hard to classify consistently, because the variables used to classify are e.g., not tangible or open for interpretation.

It is a challenge to design a classification or a full *taxonomy* without losing these properties. On top, a vulnerability taxonomy needs to have a class for any vulnerability, and should not classify non-vulnerabilities. This prompts the following research question:

> **RQ2**: What tangible and unambiguous properties does a vulnerability have?

Afterwards, we can question what these properties mean in the context of NGMEs.

1.1.2 Challenges in the Control Phase

When a vulnerability is exposed, it is often a small and limited issue, where e.g., certain input characters must or cannot be used, certain functionality is invoked and — in the case of memory issues — it almost certainly leads to a crash. It becomes the exploit writers' task to avoid crashing behaviour while expanding the small issue into an opening big enough to perform or execute a given payload.

One technique to do this is called *Heap Layout Manipulation (HLM)*, a method to force overwriting a data-structure of the exploit writer's choice. Previously, HLM was situational since the first goal of an attacker was to overwrite a control pointer and hijack the control-flow. With NGMEs however, there is a clear need for data manipulation to make the exploit work. As such, HLM is gaining traction with its use in NGMEs. This technique is strongly dependent on the actions or operations that the exploit writer can perform, and it can take large amounts of time to set up the heap in the preferred layout. Preferably, we can off-load the exploit writer with simplifying or solving the HLM task. However, the HLM problem is undecidable, so finding a solution is not guaranteed.

Furthermore, if the exploit writer has no hand in the HLM process (e.g., through an automatic solution), important details such as functions invoked or resulting memory layout may be opaque. This may lead to them reverse-engineering the resulting HLM solution, which is an unnecessary effort when automation is the goal. In other words, the HLM solution should present the exploit writer with all relevant information to complete the exploit writing process. Finally, applications can differ in heap manager usage, where different managers can behave differently — changing the problem-space again. Summarising, we question the following:

> **RQ3**: Can we find a generic solution to solving HLM problems that is explainable and heap manager agnostic?

1.1.3 Challenges in the Payload Phase

With the widespread deployment of CFI on the horizon, the attack surface for memory vulnerabilities will change drastically once again. The new NGME attacks are hard [25, 109], but it still remains possible to execute payloads. We can protect ourselves with existing tools [137, 136] but with a very large overhead. Thus, to properly protect against NGMEs we require a fast solution that still blocks the full class of attacks. Our approach is to focus on the payloads often used, and find a way to protect this functionality from attacks — even in the case of a successful control phase. We do this by observing that only a few system calls are used in these common payloads, where the goal becomes to protect the arguments to those few system calls. On top, any solution must incur limited overhead or it will not be useful. We focus on the Data-only attacks of NGMEs, leaving the CFB capability within NGMEs as future work as part of incremental research. This leads us to our final research question:

> **RQ4**: Can we protect applications against data-only attacks by blocking off common payload targets with a limited performance overhead?

In practice, this research led to an interest and understanding of heap manager behaviour and its weaknessess, which consequently inspired the research of **RQ3**.

1.2 Contributions

This work is based on a three phase paradigm, with contributions to each. We will thoroughly discuss the exploitation paradigm used in this dissertation before diving into each individual aspect (**Chapter 3**). Afterwards, we present a vulnerability taxonomy, an offensive control technique and a payload protection scheme for each phase respectively. In detail, this dissertation provides:

1. A taxonomy for vulnerabilities with a small number of classes called `GEN`. In contrast to existing taxonomies, `GEN` classifies bugs and vulnerabilities with exactly one class based on a discrepancy between two abstraction layers. `GEN` introduces two complementary concepts, Incorrectness and Undefinedness, to explain what abstraction layer is responsible for the discrepancy. `GEN` also considers the security policy as a separate abstraction layer, something overlooked by previous taxonomies. `GEN` is directly linked to the scope of impact of a vulnerability and to bug-finding techniques. (**RQ2, Chapter 4**)
2. Hack the Heap: A puzzle game that simulates heap behaviour to solve the Heap Layout Manipulation (HLM) problem. The goal of the game is to perform the Heap Layout Manipulation, i.e., set up the heap in a hacker-preferred way. A solution in the game is directly transferable into the real-world, even when the game presents a simplified version of the problem. Hack the Heap provides a visualisation and simplification of the problem-space while being heap manager-agnostic. Furthermore, it can crowdsource the exploit writer's task by letting other people play the game and find various solutions, from where the exploit writer can choose an HLM solution of choice to continue writing the exploit. (**RQ3, Chapter 5**)

3. System Call Argument Integrity (SCAI): A new paradigm for protecting common payloads. It uses the observation that common payloads use common system calls with customised argument, and protects the data-flow into these security-sensitive system calls. Furthermore, it leverages the power of heap manager behaviour to separate the arguments from the remainder of the application. SCAI uses Intel MPK technology to severely limit the runtime overhead of applications, and does not require instrumenting (dynamic) libraries or kernel modifications. SCAI protects the application from generic payloads, even in the case of a successful data-only compromise. (**RQ4, Chapter 6**)

Together, our contributions give a greater insight into the threat of NGMEs. Why they still exist; how the residual attack surface can be more effectively utilised; and what the next step is in protecting against NGMEs. We conclude the work in Chapter 7 where we discuss the above contributions in the context of **RQ1**.

Chapter 4 of this thesis is currently under revision at Elsevier Computers & Security, coauthored with Jassim Happa and Daniel O'Keeffe. Chapter 5 of this thesis has been published at WOOT 2022 [82] titled "Hack the Heap: Heap Layout Manipulation made Easy", coauthored with Daniel O'Keeffe. It passed the Artefact Evaluation with its open-source code available[1] [2]

1.3 Researching Offensive Security

Even today, offensive research is regularly regarded as unethical. At first sight, this is understandable: research results are shared with the world on how to make cyber attacks better, simpler, easier to execute or quicker to develop. As this thesis also presents offensive security research (Chapter 5), it is vital to understand why it is a necessary evil.

Sharing the information *democratises* the exploitation process. If novel attack techniques are available to anyone, it limits the effect of concurrent information in private and government instances. This may in turn make it easier to exploit new vulnerabilities/CVEs, which is the best way to determine the severity of a vul-

[1]The website is available here: `https://github.com/Usibre/hacktheheap`.
[2]The generator is available here: `https://github.com/Usibre/hacktheheap-puzzlegen`.

nerability and prioritise what vulnerabilities to patch. Typically, this is organised in bug-bounty programs where ethical hackers can get paid sums of money after submitting completed exploits. These programs provide exploit developers with an ethical way to do their job and be rewarded with it, while aiding the software development company with vulnerabilities and exploitability proofs.

Besides, most NGME attack paradigms currently being researched are not yet shown to be used in practice. These attack paradigms (e.g., data-only attacks [32, 109]) may take years before a realistic attack will be seen in the real world. The offensive research *did* lead to a plethora of protection schemes [142, 34, 180, 80, 27], attempting to have practical defences available when these attacks become realistic. If we want research to make the world a safer place within the digital domain, we need to perform offensive research to be ahead of the attacks in the wild; protect against these attacks; determine severity and bug-fix priority; and make the information available to anyone instead of using it as a secretive tool in cyber warfare.

Finally, the existing structure of Chapter 5 teaches people how to perform one aspect of exploit development to crowdsource this aspect. Our approach simplifies and teaches Heap Layout Manipulation, which could aid newer exploit writers to aid with the exploitation and in time with bug-bounty programs. However, it is only one step in the exploitation process (as Chapter 3 will highlight). On top, the crowdsourcing aspect (i.e., submitting custom puzzles for others to play) is not available to the general public.

Background 2

This chapter will provide the general background information and research related to the following chapters. We begin with a discussion of vulnerabilities as a concept and their classification. We then introduce application memory, and memory safety followed by a deep dive into the heap. After, we discuss system calls and a historical timeline on exploitation and defence techniques of memory vulnerabilities. Finally, we discuss the impact of crowd-sourcing research and gamification.

2.1 Bugs, Vulnerabilities & Weaknesses

Some mistakes in software can be abused by attackers to compromise the software itself or even its underlying system. These mistakes are generally referred to as vulnerabilities or weaknesses. The true meaning of a vulnerability is different according to different authoritative organisations. For example, FIPS200 defines a vulnerability as a “Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source” [140]. NIST CVE however defines a vulnerability as “A weakness in the computational logic (e.g.„ code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability” [130]. Among practitioners the exact definition may not be easily spelled out, but the intuitive understanding of what is and what is not a vulnerability is mostly clear. If a software (or hardware) mistake poses an opportunity for attackers, it is problematic and will be called a vulnerability.

Because software is easily copied, distributed and reused, popular implementations of software are deployed numerous times world-wide. A single vulnerability in a popular application could mean that individuals and organisations throughout the world could be at risk. In an attempt to limit these ramifications, the Common Vulnerabilities and Exposures (CVE) database was founded. The CVE keeps track of all vulnerabilities within popular software and hardware products, and labels them with unique IDs (e.g., CVE-2019-11839 [53]). The CVE ID can be linked with other meaningful information about the vulnerability, e.g., a severity score, a vulnerability type and more details/resources.

2.1.1 Vulnerability Classifications

Many attempts have been made to classify both vulnerabilities and attacks, in which certain vulnerability characteristics are key [102]. In early works, the Protection Analysis (PA) project [14] collected vulnerability samples in order to extract abstract patterns. Over 100 samples were collected and subsequently classified into four main classes: *Domain errors, validation errors, naming errors* and *serialisation errors*.

Bishop [15] used these classes as part of their UNIX vulnerability classification, calling it the *nature* of the vulnerability. Additionally, they considered other properties, most importantly the *Time of Introduction* based on earlier work by Landwehr et al. [118]. The Time of Introduction considers three phases: *development, maintenance* and *operation*. Bishop points out that ambiguity can occur if categories are not explicitly specified in a mutual exclusive manner.

Bishop's Time of Introduction was redesigned in an attack classification taxonomy by Howard [95]. To categorise attacks, Howard specifies the type of vulnerability, in which three categories arise: *implementation, design and configuration*. Nowadays, the Time of Introduction property is usually referred to these three categories.

In more recent work, we strike a rough separation between fundamental research — work that is aimed towards understanding vulnerabilities — and operational research — work that is aimed at achieving direct real-world impact. When unclear, we separate based on the approach of the project: if classes follow a bottom-up approach we label this operational, and top-down approaches are labelled fundamental. Ultimately, this separation is to aid the reader in becoming aware of the current state-of-the-art and holds no further research significance.

Fundamental Vulnerability Classifications

In 2008, Meunier summarised the different properties used in various classifications [127]. Properties such as *exploitability* and *disclosure process* measure a level of severity but can be hard to determine for a given vulnerability. A classification by *genesis* determines whether the vulnerability was intentional and if so, whether the intent was malicious. This is a difficult classification since intent and maliciousness are not objective measures. Meunier also discusses the *software development life-cycle* (SDLC) categorisation, an expansion on the Time of Introduction property from Landwehr [118]. SDLC includes a separation between integration, maintenance and requirements while keeping the main categories of design, implementation and operation. Meunier finishes with a set of 6 criteria (4 defined and 2 implicit) a taxonomy requires. We refer the reader to Meunier for a full overview of the top-down properties used for classification in previous work.

The taxonomy criteria of Meunier have been expanded upon by Derbyshire et al. [63]. They examine end-to-end *attack taxonomies* rather than standalone vulnerabilities, and analyse 7 attack taxonomies from both theoretical and practical perspectives. In total, 10 different theoretical criteria are outlined, together with a separate test to determine use in practice.

Simmons et al. proposed AVOIDIT, an attack taxonomy aimed towards educating the defender [169]. AVOIDIT classifies attacks along four dimensions (i) Attack Vector, (ii) Operational Impact, (iii) Defence, and (iv) Information Impact. Although AVOIDIT is not a vulnerability taxonomy, its attack vector dimension resembles a vulnerability classification. AVOIDIT defines ten different categories for the attack vector, including buffer overflow, file descriptor, input validation and social engineering. Unfortunately, this classification has limitations since a single vulnerability can fall into multiple classes, i.e., classes are not mutually exclusive. For example, buffer overflows often occur due to a lack of input validation. The authors do mention that an attack could be comprised of a chain of attack vectors, but this should not lead to ambiguity. According to Derbyshire et al. [63], AVOIDIT only covers two of their ten criteria.

More recently, Li et al. proposed a model based on the complexity of abusing a vulnerability [124]. This is a hard problem to assess, but they attempt to capture the complexity with a few broad categories. If the vulnerability results in control over resource usage on the target system (e.g., potentially causing a Denial-of-Service (DoS) attack), they refer to it as an Aging-Related Vulnerability. If the vulnerability is dependent on external factors such as timing or the environment, it is called a Non-Aging-related Mandel Vulnerability. Alternatively, when a fixed set of conditions is required for the vulnerability to be exploited, it is referred to as a Bohr-Vulnerability. When a vulnerability matches none of the above, they consider a catch-all Unknown Vulnerability category.

Surprisingly, an integer overflow issue falls in the category of the Aging-Related Vulnerability according to the authors. This type is clearly different from other vulnerabilities in this category, which only deal with DoS attacks, and might potentially aid the exploitation of a separate vulnerability (e.g., in HLM [92]). An integer overflow instead can lead to a complete compromise of the internal state of a process. In fact, a buffer overflow attack is mapped to all three categories ([124], Figure 3). Following the criteria from Derbyshire et al. [63], this approach lacks (i) mutual exclusiveness, is ambiguous and does not represent socio-technical attacks among others. Instead of their proposed use-case of understanding how vulnerabilities can be exploited — which one requires to use the classification — this methodology might be of more use in an operational vulnerability assessment setting.

Operational Vulnerability Classifications

The Seven Pernicious Kingdoms by Tsipenyuk et al. [190] classify vulnerabilities into "7+1" different categories. The first seven are: (1) input validation and representation, (2) API abuse, (3) security features, (4) time and state, (5) errors, (6) code quality and (7) encapsulation. According to the authors, all the above can be identified and resolved within the source code ("implementation" category in the Time of Introduction/SDLC). Finally, their last class is an environment class that refers to issues with configuration and the environment in which the application is run. As the authors acknowledge, it is clear that the classes are not mutually exclusive. For example, a failure on input validation within an API can create a vulnerability, and a buffer overflow can already occur in classes 1, 2 and 6. Furthermore, a category such as code quality is inherently subjective and can easily be used as a catch-all for software bugs. Ultimately, their goal is to "help software developers understand the kinds of errors that have an impact on security". Within the Time of Introduction classification, a design category is absent.

MITRE OVAL. Common in an operational setting is the MITRE OVAL standard [3]. The OVAL standard is a collection of various classifications and properties of flaws, vulnerabilities, exposures and weaknesses, whichever word more closely resembles the situation. Among others, OVAL includes the known CVE collection and ATT&CK, but also the lesser known CWE standard and CAPEC. Every piece of OVAL is focused towards a different goal: Where CVE is operational by nature and contains common vulnerabilities across products, CWE aims to classify weaknesses in a stratified framework, aiming to list all potential issues that could give rise to a vulnerability. Note that CVE is not a classification model (in contrast to most OVAL efforts), but rather an operational framework for listing vulnerabilities. Not part of MITRE OVAL, the CVSS aims to measure vulnerability severity in a stratified manner.

The Common Attack Pattern Enumeration and Classification (CAPEC) [10] is a taxonomy of attacks, aiming to aid the entire software development lifecycle (SDLC). CAPEC is a tool that aids testing, analysis and threat modelling. It is organised into two independent tree structures with 575 nodes at the time of writing.

The first tree is "Mechanisms of Attacks". Each category here represents a different mechanism that could be used in the attack of a system. An example category here is "Abuse existing functionality", that contains "API Manipulation" as subcategory among others. Alternatively, the second tree-structure on the nodes is "Domains of Attacks", with categories such as *software* or *supply chain*. Here, API manipulation is a subcategory of software instead. All nodes contain a short description of the execution flow of a potential attack.

The attack categorisation as listed are mainly operational in nature. The number of categories and attack patterns is large and attacks can fit in more than one category. Furthermore, they can represent completely different stages of the same attacks. For example, fuzzing is listed as a type of attack whereas this has a completely different stage of an attack compared to code injection: fuzzing may *find* a code injection opportunity. It is disputed whether fuzzing is indeed an attack vector at all — and raises the question why CAPEC does not contain a complete

[3]https://oval.mitre.org/

bug-finding technique list if so. It would be interesting to have a relation based on the attack stage and related stages (such as fuzzing flowing into a code injection attack).

The Common Weakness Enumeration (CWE) [133] also has a bottom-up approach but focuses on vulnerabilities rather than attacks. This is a big difference: Where CAPEC aims to categorise attacks from an attacker perspective, CWE tries to answer the question what the issue is with a given application. This line gets blurred in some types of weaknesses and attacks. For example, CAPEC has an *Overflow buffer* attack category whereas CWE has a *Buffer Copy without Checking Size of Input* category.

Like CAPEC, CWE has seen an explosion in categories from its bottom-up approach, making it difficult to use for classification. For example we can look at `CVE-2019-16724`, a vulnerability in File Sharing Wizard 1.5.0 where part of the input data is not properly validated (CWE-20) and triggers a buffer overflow (CWE-118, CWE-119, CWE-120, CWE-788) that leads to an out-of-bounds write (CWE-787). This out-of-bounds write can be abused to create a "write-what-where" primitive (CWE-123). An exploit is known in the form of a Metasploit module[4] that injects a shellcode (CWE-94, CWE-96).

In fact, `CVE-2019-16724` is classified to CWE-120 and whilst its CVE states that the issue is similar to `CVE-2010-2330` and `CVE-2010-2331` — which are both classified as CWE-119. Thus while CWE can excel in giving a solid explanation to the issue of a known vulnerability, it can be highly ambiguous as a vulnerability classification.

Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) is a model and knowledge base to identify security gaps in organisations, and addresses them prior to adversary attacks. ATT&CK organises attacker behaviour into technical objectives that adversaries want to achieve, such as data collection, discovery, defence evasion and exfiltration among others. Each objective (called a tactic in ATT&CK terminology) contains a list of concrete techniques. For example, the collection tactic contains audio capture; data from local system; email; and video capture. The specific technique depends on the vulnerabilities present on the attack surface, and the capacity and capabilities of the attacker. Each technique provides

[4]https://www.exploit-db.com/exploits/47412

a description of the method, the systems it applies to, what adversary groups use such method (if known), as well as any mitigation/remediation actions. The model takes a form that resembles a set of 2D matrices, including: (1) preparatory tactics and techniques (akin to reconnaissance and weaponisation in the cyber kill chain [99]), as well as (2) Enterprise and (3) Mobile Matrices. The model is intended for security practitioners to identify potential vulnerabilities in existing digital platforms and to address them directly. The model is not mainly used for addressing software vulnerabilities, but rather (whole) systems deployed in organisations.

Common Vulnerability Scoring System (CVSS) [126, 162] is a standard measurement system to score vulnerability *impact*. It is composed of three metric groups: base, temporal, and environmental scores. These three rely on consistency of metrics most of which are represented as a "low–high" range. The base score represents the foundational characteristics about a vulnerability that are constant, irrespective of time and user environments. The base score is the mandatory score for assessing any vulnerability when using the CVSS standard. It identifies the access vector, access complexity, privileges required, user interaction, scope and Confidentiality/Integrity/Availability (CIA) impacts.

The temporal score identifies the characteristics of a vulnerability that change over time but not among user environments, such as: to what degree is it exploitable, what remediations exist and the confidence analysts have in the vulnerability reports. Finally, the environmental score reflects the characteristics of the environment, including potential mitigation techniques that may limit the impact of successful exploitation and additional assets available after exploitation. When the metrics are assigned values, the underlying equations can calculate a final score ranging from 0 to 10.

Weird Machines

Once the vulnerability is triggered (i.e., reached/executed), the applications' logic is compromised and "weird" things can happen within the execution. According to the Langsec community, we enter a *weird machine* [20, 9, 197]. The weird machine is a conceptual state machine model consisting of all states that were not supposed to exist. When a vulnerability is triggered, the weird machine is entered and its states are exposed. The weird machine technically expresses an undocumented program with unknown semantics [70]. *Programming* the weird machine becomes analogous to writing an exploit.

Attempts have been made to formalise weird machines. Dullien proposed the usage of a state automaton to represent weird machines [70]. An approach by Vanegue uses abstraction levels [197], analogous to our approach in Chapter 4. D'Silva et al. extended this approach from Vanegue, in the context of compiler optimisations that nullify security properties [69]. They conclude that different levels of abstractions are required to reason about low-level behaviour of code. They show the abstraction level used to reason about optimisation may overlook issues that arise on a different level of abstraction.

Also focusing on weird machines on the level of compilation are Paykin et al. [150]. They consider the source code as ground truth. Anything that deviates from the source code when executing the compiled program is seen as a weird machine. They take a formal approach to model this using programming language techniques. Any deviation to the source code is called an exploit. As such, their formalism captures vulnerabilities as a result of the compilation process and beyond.

Limitations of the Weird Machine Model. One paper by Erik Poll discusses forwarding flaws: issues that appear when data is passed from one application to another, e.g., SQL injection. He discusses how this is a separate issue compared to processing issues [153]. Specifically, Poll mentions that forwarding flaws are different by nature as they are deliberately introduced, but accidentally exposed. Through this, it is concluded that the forwarding flaws do not expose weird machines.

Although forwarding flaws clearly are fundamentally different, we argue that it *does* expose a weird machine. Lets take the example of a search engine that fails to escape quotemarks(") when forwarding a query to an SQL database. Using one or more quotemarks inside the search query will *not* search for the exact string, as the quotemarks are interpreted in the context of the SQL code. The deviation occurs in the application, as the semantics of the resulting SQL query already deviates from its intended semantics. In other words, the application behaviour deviates from the intended behaviour, so a weird machine is present in the application.

Rather, the issue is two-fold. First off, the deviation itself is expressed in data rather than code. Although this is in line with the Langsec principles, its deviation is not present when representing the weird machine as state machine — the common representation of weird machines. The second problem lies in the decoupling from the original application and its weird machine(s). One vulnerability can express many weird machines, depending on the state of the application at the moment that the vulnerability is expressed. As such, it is the exploit writers' goal to "choose" or set up the right weird machine to make it possible or easier to program the required exploit. In the SQL example above, the vulnerability is expressed with any input string that contains at least one quotemark. Yet, the small weird machine it creates is highly dependent on the semantics of this string in the context of the application it is forwarded to. Exploitation writing is thus not only programming the weird machine, but also choosing the right point of entry from the vulnerability, i.e., choosing the right weird machine to program.

A separate issue of the weird machine model is when multiple vulnerabilities are used within a single exploit. Especially when vulnerabilities of different abstraction levels are leveraged, the weird machine model cannot contain this level of information within a single representation: a logic bug is not a vulnerability when looking into the weird machine concept in bytecode fashion like Dullien [70] as it lacks high-level semantics. If the weird machine however models the logic operations of the application, a subsequent memory violation cannot be modelled within the same weird machine.

This is not to say that the weird machine model is not accurate or useful. Instead, we argue that the weird machine model is in its infancy and needs to become part of a larger model. Only through a larger model (with e.g., the potential for multiple weird machines or abstractions) can it fully grasp the full breath of vulnerabilities in the context of exploit writing.

2.2 Memory

To fully understand memory vulnerabilities, we require a good understanding of how memory is organised within an application, what is meant with memory (un)safety and how we can protect this memory. These are all discussed in this

```c
int main() {
  char name[128];
  scanf("%s", name);
  printf("Hello, %s!", name);
  return 0;
}
```

Listing 2.1: Example C code containing a stack-based buffer overflow.

section.

An application process requires memory to perform necessary operations. Although physical RAM is used, the Operating System (OS) reorganises the available memory into *virtual memory*. We will not go into detail about the inner workings of the OS and its memory management, but it is important to understand that a memory address does not correspond to the hardware address of a running application. When starting an application, it starts an application *process*. The process gets usable memory in the same address range: accessing this memory by an application gets translated by the OS to its respective hardware address.

Because it gets translated, any memory violation within an application process cannot write outside the boundaries of the application itself. This provides the machine with a vital piece of protection, as one vulnerable application does not directly put other processes in danger. This does not mean that a memory violation is benign, as it can still affect the internal structures of the application. From hereon, "memory" refers to the virtual memory as used by a single application process.

2.2.1 Application Memory

Within an application process, memory is used for various purposes. Memory is generally executable (i.e., for executing the code), writable (e.g., for managing data) or read-only (for unchanged data). Writable memory can be divided again into 4 different groups: global memory, the stack, the heap and raw memory requests.

The *global* memory is of a static size and will exist for the entire duration of the application runtime. Secondly, applications use a *stack* containing local variables and buffers. This memory is linked to a function: when the function finishes, these variables automatically go out of scope. Thirdly, we have dynamic memory called *heap* memory. The size and lifetime of heap memory is under control of the developer.

When heap memory is required, a function can be called (e.g., `malloc()`) to request a buffer of a given size. At the end of its usage, the developer is responsible for returning the memory. This is done with a call to `free()`. The heap is discussed further in Section 2.3. Finally, large chunks of unmanaged memory can be requested *manually* (e.g., `mmap` in unix-based systems). This provides the developer one or more pages of raw memory. If this large chunk is used for more than one data structure, the developer is responsible for managing the layout. A typical memory layout in an application is visualised in Figure 2.1.

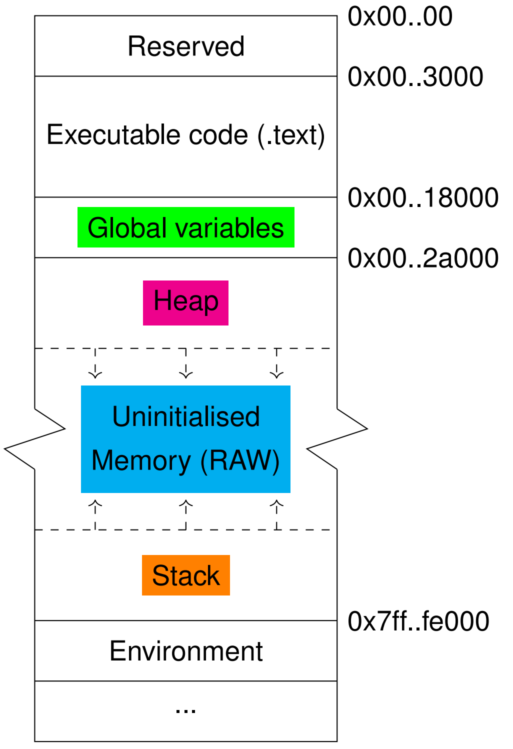


Figure 2.1: A typical memory layout of a *NIX application.

2.2.2 Memory Safety

Some programming languages (e.g., C, C++ or more recently “unsafe” Rust) do not enforce memory boundaries. This is generally faster compared to an automatic system like Garbage Collection or runtime checks [65], but a mistake by a developer could mean that memory boundary properties are violated. These programming languages are therefore considered *memory unsafe*. In order to gain memory safety in languages, we need to combine *spatial memory safety* and *temporal memory safety*.

Spatial Memory Safety means that no memory buffer access — when accessed for both reading and writing — can end up outside the spatial boundaries of said buffer. As an example, there is a spatial memory safety issue in Listing 2.1. Under normal circumstances, someone’s name does not exceed 127 characters[5]and the program would execute as intended. Any input that exceeds 127 characters will

[5]127 characters not including the null delimiter as 128th character.

```
 1  [..]
 2  int main() {
 3    int *number1 = (int) malloc(sizeof(int);
 4    scanf("%d", number1);
 5    free(number1);
 6    int *number2 = (int) malloc(sizeof(int));
 7    scanf("%d", number2);
 8    printf("%d + %d = %d!", number1, number2, number1+number2);
 9    free(number2);
10    return 0;
11  }
```

Listing 2.2: Example C code containing a heap-based use-after-free.

write outside of the allocated "`name`" buffer. This could overwrite other data saved on the stack or metadata saved on the stack. Spatial memory safety also includes underflows or arbitrary reads/writes, as can be demonstrated with e.g., a format string attack.

Temporal Memory Safety is instead concerned with the *lifetime* of memory buffers. This occurs when the memory is not ready to be used yet/anymore. For example, when a variable is read before initialisation; is read or written to after its lifetime; ending the lifetime of memory more than once; etc. A common mistake w.r.t. temporal memory safety is a *use-after-free* (UaF), an example of which is shown in Listing 2.2. In this example, `number1` is freed on line 5. The lifetime of the `number1` variable has ended, but interactions still occur with this variable on line 8. Depending on the heap manager used, the `number2` variable here can be placed in the same location as `number1`. This would cause writing to `number1` to overwrite the value of `number2`. Heap manager behaviour is discussed later in this chapter.

2.2.3 Memory Protection

Memory pages in an application have three protection flags, determining whether the memory is (1) readable; (2) writable; and (3) executable. In Section 2.2.1 we discussed all writable sections. However, runtime requests can change this: on Linux systems this is done with the `mprotect` system call. This can set any of the above flags to true or false, making any combination possible[6]. This has been used to create execute-only memory [163], which prevents read-outs of the memory and

[6]Some combinations of memory protection flags may not be supported by all kernels.

thereby enforces confidentiality (e.g., to protect intellectual property and hinder code-reuse attacks). The same mechanism protects against attackers overwriting non-writable data (e.g., the bytecode itself).

Flags set on memory pages are stored in the kernel. Per process, a limited number of contiguous memory regions can have differing flags according to the `mprotect` specification. If the application contains too many different regions (differing in protection flags), a consecutive call to `mprotect` can fail.[7]The protection flags are shared across different threads.

Information Hiding. Protecting memory against adversaries has become increasingly popular in the last decade [116, 23, 43, 172, 27, 80, 142, 139]. An `mprotect` protection can only cover a small amount of differing flags and switches slowly, so alternative solutions have been attempted. For example, Information Hiding (IH) allocates a small amount of memory in a randomised location [116]. Given the large space for virtual memory (128TB), guessing a small memory area becomes difficult enough to provide guarantees similar to guessing encryption keys. The used memory is typically surrounded by guard pages, where any read or write will trigger an exception. However, IH relies heavily on not leaking the hidden memory address in any way for an exploit writer to find. This has been shown to be more difficult than previously expected [43, 78, 85, 144]. Regardless of the scrutiny surrounding information hiding, the LLVM CFI implementation relies on IH at the time of writing [185].

Software Fault Isolation. An alternative solution masks pointers before using them. This is known as Software Fault Isolation (SFI) [201, 165]. Before using any pointers, a bitwise `and` can be used to force certain bits to 0, and conversely a bitwise `or` can set bits to 1. With memory segregation, this can force pointers to be in or out of a specific region. Afterwards, the application can either continue with the masked pointer, or compare the value against its original (and throw an exception if the check failed). For SFI to be fail-safe, the memory region needs to be determined at compile-time to ensure that an attacker cannot tamper with the pointer masks. This also requires the maximum memory *size* to be determined while compiling.

[7]Typically around 16 different regions, including the pre-existing regions where e.g., code can be executed; data can be edited but not executed; or read-only memory that can do neither.

For sandboxing, SFI can be used on specific parts of code (e.g., plugins or user-supplied code). For the opposite — protecting a specific memory area — every single memory read or write instruction will need to be masked for confidentiality or integrity respectively. This can lead to a substantial performance overhead. Besides, SFI does not protect uninstrumented code in contrast to the above techniques.

Intel MPK. The CPU manufacturing company Intel responded to the need for additional fast memory protection as well. They developed a hardware solution inside CPUs called Memory Protection Keys (MPK). These are protected protection bits (for reading and writing) that can protect up to 16 different memory domains that can be set through custom instructions. MPK enforces read and write protection at the hardware level on a per-thread basis, with quick user-space interaction.

2.3 The Heap

The heap is a flexible memory region used by applications for data with a custom lifetime. It can be interacted with through the malloc API as specified in C18, also known as C17 or ISO/IEC 9899:2018 [108], in section 7.22.3. The standard describes how to request a memory chunk (through `malloc`, `calloc`, `aligned_alloc`, resize an existing memory chunk (`realloc`) or return a memory chunk (`free`). An API implementation of the heap is called the *heap manager*.

The specification does not describe *how* the memory is managed internally: this is to the discretion of the implementation. How to distribute pieces of a large contiguous chunk of memory with a limited performance and memory overhead poses various challenges. Generally, different heap managers trade off space efficiency for memory overhead or security guarantees. We discuss the most significant design decisions below.

Fitting Methods. When a memory request is received by the heap manager, it is the task of the heap manager to quickly and efficiently return a memory region of (at least) the requested size. This memory is selected from one contiguous region in most cases. Yet, previously allocated memory chunks can be freed or resized at any time, which can fragment the heap memory area quickly. The most fundamental question on a request for a memory chunk, is how to search for a large enough chunk of available memory: this is called the fitting technique or fitting algorithm.

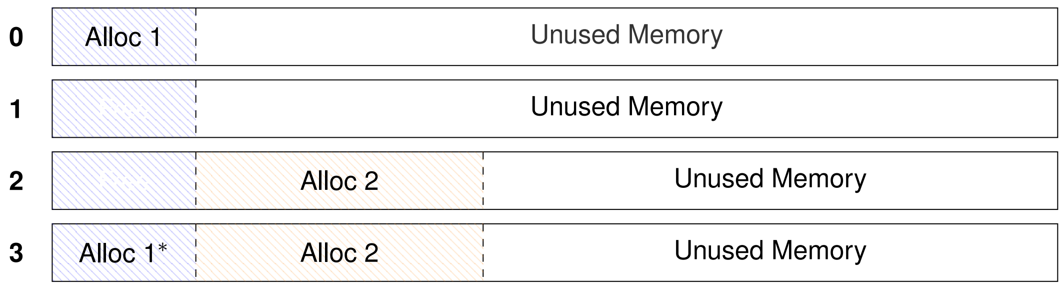


Figure 2.2: Here we show the behaviour of free-lists. When an object is freed, for example freeing '`Alloc 1`' (0 $\rightarrow$ 1), its location remains preserved for another memory request of this size. If we request a different size chunk '`Alloc 2`' (1 $\rightarrow$ 2), it will not use this memory since it is reserved. When this size chunk is requested again by requesting '`Alloc 1*`' (2 $\rightarrow$ 3) it will fill the spot with either the LIFO or FIFO principle (typically LIFO, stack-based). With segregated memory, a number of locations are preserved per bin size *a priori*.

One such fitting technique is "First Fit". Here, the heap manager starts searching from the start of the heap memory region. We traverse to the end, checking all free regions. As soon as we find a memory chunk that is big enough to hold the request, we take the start of this memory region. Upon freeing the chunk, we may require *coalescing*, merging the freed area with adjacent free areas to create one bigger free area. If we don't, we may end up with many tiny (and virtually unusable) free memory chunks, none of which are big enough for larger requests.[8]

Other fitting techniques include "Next Fit" (where we start searching where we left at the last search), "Best Fit" (where we search the entire area for the smallest fitting empty region, gaining efficient memory usage at the cost of performance), and "Free Lists". With Free Lists, we group requests in different sizes, padding each request to these sizes. Each size has an associated free list containing available memory chunks of that size. If nothing is available, we use unsorted memory to return a memory chunk. When this heap chunk gets freed eventually, it is added to the free list to be reused for other memory requests of the same size. A visual example on the Free List mechanism is shown in Figure 2.2. Free lists are common in heap managers, although significant differences lie in (1) whether to adapt a LIFO (Last-In-First-Out) or FIFO (First-In-First-Out) policy, (2) freelist sizes and ranges between sizes, (3) coalescing, and (4) segregation as discussed below.

[8]A visual demonstration of First Fit can be found on https://hacktheheap.io/game.htm?id=1 using the website from Chapter 5.

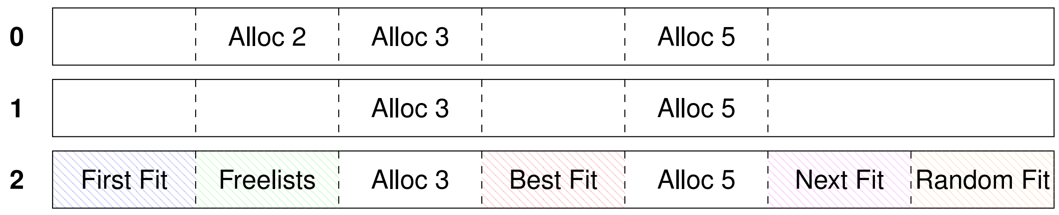


Figure 2.3: This figure shows the different locations where a heap manager can place a subsequent request. Given the heap state as marked in the first bar, "Alloc 2" is freed which results in the second bar (1). After this, the different marked area shows the different locations depending on what fitting technique used by the heap manager. Note that all allocations in this example are of equal size. Random fit could technically occupy any of the available spots, but has the potential to land on the marked spot.

The difference between different fitting techniques is visualised in Figure 2.3. Here, we allocate 5 equally sized memory chunks before freeing the first and fourth allocation. The resulting memory layout is shown in the State 0. Following the steps in the figure, we free the second allocation, resulting in State 1. The next allocation can occur in a large amount of locations, depending on the fitting technique deployed, as State 2 shows.[9]

Note that the fitting techniques as discussed are theoretical models. While heap managers generally follow one or more algorithms as described above, additional deviations from the model algorithms exist to enhance performance or lower fragmentation.

Segregation. When using free lists, we *could* pre-allocate a multitude of memory chunks of each size. Doing so enforces that every allocation of the same padded size would be in the same area until full. Freelists for each size can be pre-filled after pre-allocating. If the heap manager runs out of space for a given size, it can either split a larger existing block for new entries (typically referred to as "Segregated Fit") or take an unorganised chunk and split this into entries of the requested size (called "segregated storage") [205]. Note that the use of free lists does *not require* any form of segregation — Figure 2.2 for example.

Non-determinism. When heap managers become predictable, their behaviour can be abused in the existence of a memory vulnerability. *How* to leverage the predictable behaviour is studied in Chapter 5, but some heap managers have taken

[9]Random fit could differ more, depending on its implementation.

this into account and added direct non-determinism within the fitting method. This needs to be done carefully to avoid excessive fragmentation without losing non-determinism. For example, the Windows Low Fragmentation Heap (LFH) does so by choosing a random entry from a pre-allocated freelist. Yet, attempts to limit the heap fragmentation have been shown to become deterministic enough to abuse in the past [193, 159], e.g., through heap spraying [66].

Metadata Protection. Another design decision is whether to include metadata adjacent to the returned memory (e.g., tcmalloc) or in a separate metadata region for all metadata (e.g., LFH). Besides non-determinism, modern heap managers have incorporated security mechanisms to protect metadata. Metadata is well-known to enable various attacks [206], e.g., an unsafe unlink attack [155]. To protect heap metadata, it can be encoded, and its location can be randomised and surrounded by guard pages. Illegal metadata behaviour (e.g., double free) can be detected with additional checks and metadata can be heuristically checked to be reasonably correct (e.g., freelist pointers pointing into the heap region). Heaphopper [71] can be used to check heap metadata attack safety, and a safe allocator can already be deployed [141].

Size Thresholds. Heap managers change their internal behaviour depending on the size of the request. This is good, as small requests are allowed to be less space-efficient without losing too much effective memory, speeding up the request process. Similarly, when requests become very large, heap managers will often request the memory from the OS (i.e., through the `mmap` system call in Linux). A number of thresholds are determined per heap manager where each range of sizes adapts different behaviour. For example, ptmalloc2 uses single-linked lists when the requested size is smaller than 113[10](becoming up to 128 bytes including metadata), and will call `mmap` if the request size exceeds 131040 bytes (max size of `0x1FFF0` including metadata). Different heap managers have different default thresholds.

[10]This is empirically tested and configurable.

```c
typedef struct {
  size_t id;
  uint16_t content_length;
  char *value;
} record;

record *new(size_t id, char *password,
    uint16_t length) {
  record *rec = malloc(sizeof(record));
  rec->id = id;
  rec->content_length = 2*length;
  rec->value = malloc(rec->content_length);
  strlcpy(rec->value, password, length);
  return rec;
}

void alter_value(record *rec, char *password,
    uint16_t length) {
  if (rec->content_length < length) {
    free(rec->value);
    rec->content_length = length;
    rec->value = malloc(rec->content_length);
  }
  strlcpy(rec->value, password, length);
}

void print_value(record *rec) {
  printf("Password for id %lu: '%s'",
      rec->id, rec->value);
}
```

Listing 2.3: Vulnerable example code where HLM can be used

2.3.1 Heap Layout Manipulation

The Heap Layout Manipulation (HLM) problem, also known as Heap feng shui [173, 141] or heap massaging [86], is the challenge to change the heap layout in a format that is beneficial to the exploit writer. The aim is to set up a heap layout such that the vulnerability will overwrite a memory chunk of interest. We call these chunks *targets*. Similarly, we have access to memory chunks that write outside the temporal or spatial boundaries. This can result from a temporal vulnerability (e.g., a use-after-free) or a spatial vulnerability (e.g., an overflow). We call the vulnerable memory chunk(s) *bugged*.

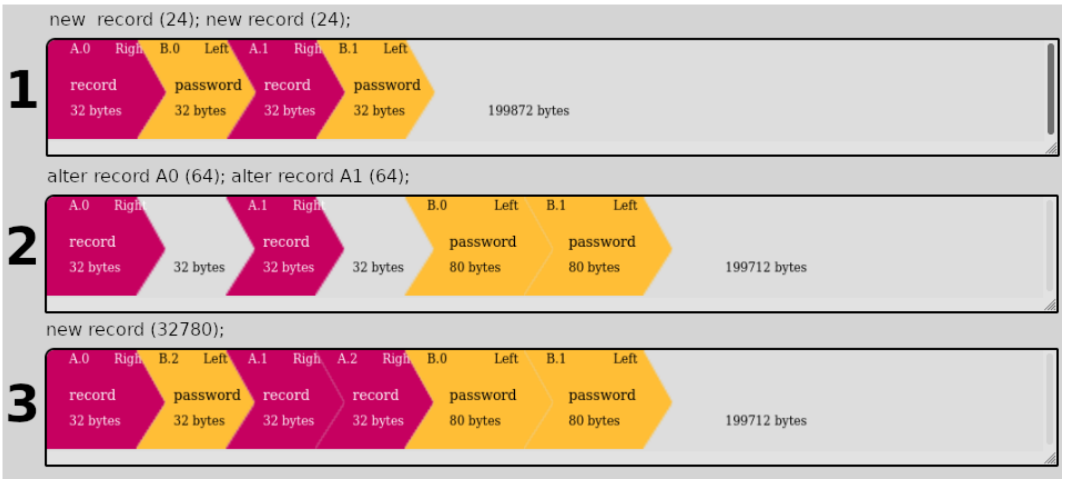


Figure 2.4: A visualisation of an HLM solution as discussed in Section 2.3.1. Between each step, code is printed which turns the upper layout into the layout below the text. In step 3, we are left with a "password" memory chunk (the left-most one) right next to a "record" memory chunk. An overflow from the password chunk would leak into the record chunk, overwriting a data pointer. The size of each puzzle piece is slightly bigger (e.g., 32 bytes) than the requested size (24 bytes) due to the added metadata from the heap manager.

Example

Consider the code in Listing 2.3. This is a simple (and poorly coded) password manager with three operations. The `new` operation saves a new record containing a password to be saved; `alter_value` can change the saved password of a record; and `print` shows the saved password on the screen. The `new` function requires a password length as well as the password. This length is checked in the copying function on line 13. Yet, the size of this new buffer is set to twice the length as seen in line 11. This creates the potential for an (unsigned) integer overflow where $2 \times \texttt{length} < \texttt{length}$ and hence the buffer would be too small for the submitted password. We refer to this allocation (on line 12) as a *bugged* allocation. Next, we need to find a useful value to overwrite with this overflow. No function/control pointers are available on the heap, but we do have one data pointer available: the `value` inside the record struct on line 4. If we overwrite this value, we can control both *where* to write and *what* to write by using the `alter_value` operation on the overwritten record afterwards. The allocation to the record struct creates this pointer on the heap (line 9), and thus we call this a *target* allocation. With the information above, the goal is to allocate a *bugged* allocation adjacent to a *target* allocation, on a lower address (so the overflow writes into the target allocation). *This is the heap layout manipulation problem*. A solution to our example is depicted in Figure 2.4 and is comprised of the following operations:

1. `new` operation with size 24, twice. Shown in layout 1 of Figure 2.4.
2. `alter_value` operations on step 1 in reverse order to size 64, as shown in layout 2 of Figure 2.4.
3. `new` operation with size 32780, that will overflow in an allocation of size 24 and write into the second record created in step 1, see layout 3 in Figure 2.4. Any additional data written to this overflow will write into the record object, overwriting a data pointer and creating an arbitrary write.

2.3.2 Existing HLM Solutions

Heap layout manipulation is gaining traction in the research community. SHRIKE [92] performs a random search for a solution, but is limited to interpreters and requires (re)compilation of the application. In follow-up work, Gollum [93] uses an evolutionary algorithm to enhance its success rate. SLAKE [33] performs heap layout manipulation on the kernel, using the kernel's heap management properties to implement a custom search technique. Similar to SHRIKE and Gollum, it requires recompilation and is limited to a narrow set of applications.

Most recently, MAZE [203] proposes a custom "dig&fill" algorithm for the HLM problem. MAZE analyses operation-based applications using symbolic execution to automatically find the operations one can perform. It then probes the heap manager with similar (concrete) heap actions to find patterns in the operations: creating or filling "holes". This converts the problem into a list of constraints, to be solved with a constraint solver. Although MAZE does not appear to require access to source code[11], it still has a number of drawbacks. First, MAZE is limited to applications that can be reasonably symbolically executed, but symbolic execution is strongly affected by state explosion and at times, imprecision. Second, the dig&fill algorithm is closely tied to the inner workings of dlmalloc and ptmalloc (a threaded

[11]MAZE's implementation is not available at the time of writing.

adaptation of dlmalloc). Finally, translation of operations into constraints can only occur if operations obey certain conditions with respect to the size of allocations. To ensure this, MAZE employs different strategies (e.g., operation merging) that while convenient for the solver still limit the search space, potentially excluding solutions.

In contrast, in Chapter 5 we propose a puzzle game to visualise and simplify the HLM problem space. The puzzle game is heap manager agnostic and although not fully automatic, it provides the exploit writer with one or more explainable, visual solutions to the problem.

2.4 System Calls

In order for user-space applications to have *any* impact on the machine it is running, it needs to perform system calls (also known as `syscalls`). The collection of system calls can be considered an API to gain any form of input or output, that works by triggering an interrupt. Then, the kernel handles the API request before returning to the user-space application with the result of the system call.

Which system call is requested is determined by the `rax` register value in Linux-based OSes, where over 350 different system calls provide an extensive list of interactions for applications [189]. Additional registers (typically `rdi, rsi, rdx,` etc.) can contain arguments to the system call (e.g., where to write input). Note that this assumes the System V calling convention for AMD64, as is the focus for this dissertation.

For example, to write to the screen in a Linux-based bash one needs the `write` system call. On an x86-64 cpu, the system call needs to be triggered with an `rax` value of 1 and an `rdi` value of 1, being standard output as first argument. Then, `rsi` needs to hold a pointer to the string that needs to be printed and `rdx` requires the number of bytes to write. Now, all we need to do is call the `syscall` opcode (analogous to the `int 0x80` on 32-bit x86 machines) and the kernel will print our message for us. Then, it will save the number of bytes written into the `rax` register as return value and return back to the application.

If an attacker tries to change the behaviour of an application to their benefit, they require one or more system calls with the right arguments to produce any beneficial output too. Instead of looking for executing a predetermined piece of code (e.g., a shellcode), we can frame the problem as executing a series of system calls (or just one) with the right arguments.

2.5 Memory Attacks and Defences: a Timeline

The concept of taking control over an application in the presence of a memory vulnerability is *at least* known from 1972 [6] and is still a real threat today [49, 50, 111]. The process has changed completely however, due to protection mechanisms in place today and possibly more tomorrow. Here we discuss the most impactful changes of the past 50 years in research and practice. It starts with an extended version of the work by Szekeres et al. [182], with a slightly different focus given todays knowledge. Afterwards, we discuss what has happened since their work in 2013.

Attack **The Start: Stack Overflows.** Anderson discussed the potential of a buffer overflow in 1972, stating "[a spatial memory vulnerability] can be used to inject code into the monitor that will permit the user to seize control of the machine" [6]. In 1988 it became clear that stack overflows were a real threat, as it was used by the notorious Morris worm to infect targets [175]. The technique however became popular amongst hackers when an article appeared in the e-magazine *Phrack* called "Smashing the Stack for Fun and Profit" [4] in 1996. Here, Aleph One discusses in detail how the return address of a stack frame can be overwritten to execute a *shell code*. Originally, a shell code is a minimum example in assembly or bytecode to open a shell. Nowadays, a shell code refers to a small snippet of bytecode the attacker wants to execute on the target.

Hitting the exact start of the shell code upon jumping proved problematic, as he stated: *"Trying to guess the offset even while knowing where the beginning of the stack lives is nearly impossible"*. If execution of the shellcode starts at the wrong position, we could miss vital instructions or execute misaligned instructions (i.e., jumping halfway into a multi-byte instruction and interpreting the instructions as different ones). Aleph One suggested the usage of `nop` instructions. A `nop` instruction is a single-byte instruction that performs the "no-operation" by doing nothing and can be added at the start of the shellcode in large amounts. Starting the shellcode execution anywhere within a range of `nop` instructions (`nop`s) will continue doing nothing until the effective code is executed without alignment issues in x86 or x86-64 architectures. Nowadays, this technique is known as a *nop sled*.

Defence **The Response: $W \bigoplus X$.** In a response to the stack overflow attacks above, various different techniques were presented to counter an easy stack overflow exploit. SPARC machines running Solaris 2.6+ were among the first to implement $W \bigoplus X$ in 1997 [184], a design principle where memory is allowed to be either **w**ritable or executable but not both[12]. This simple property single-handily stops the usage of shell codes, as the attacker injects code into a writable segment —- which would then be non-executable.

This required a fundamental change to memory, to be implemented in such a way that the OS/hardware does not have to perform additional checks upon every memory write and instruction execution, significantly slowing down the application. Multiple solutions appeared, including but not limited to the Non-eXecute bit (NX bit) and Data Execution Prevention (DEP). As of 2004 approximately, the non-executable memory property is enforced in hardware in most processors (e.g., AMD and Intel [186]). Nowadays, this is a default option on machines, including phones. Where it is not enforced through hardware it can always be enforced through the operating system or hardware enforcement can be simulated in a virtualised environment, even if less relevant nowadays.

Attack **The Alternative: return-into-libc.** Hackers were quick with a response. As early as 1997, the first exploit was written that bypasses the NX bit [171]. The attack used the code of the application *against itself* by calling the `system()` function in libc. After all, libc would always load on every application in the same memory location. The arguments to `system()` were carefully prepared in the exploit, so `/bin/sh` could be executed and a shell would spawn. This type of attack became known as a return-into-libc exploit, and its concept was broadened by Nergal in Phrack in 2001 [138].

Defence **Into the Caves: Stack Canaries.** In 1998, Cowan et al. presented StackGuard, a compile-time toolset that adds protection against stack-based overflows [47]. Here, they presented a new technique called a canary word, nowadays known as a stack

[12]Technically, this technique started off as a Non-Executable Stack specifically, later to be generalised to any writable memory.

canary. The technique is named after the Welsh miner's canary which was used to detect high amounts of carbon monoxide inside mining caves.

A canary is a word-sized random value to be placed at the start of a stack frame, between the saved base pointer and the local variables of that stack frame. Upon returning from the function, the code will check if the stack canary is still intact (i.e., contains the original value) and halt execution if not. If an attacker wants to overwrite the return address, it will overwrite the stack canary and the attack will be detected before the malicious payload can be executed. Attackers will need to skip the canary while writing out of bounds, read out the canary first and write the same exact value, or guess, or bruteforce its value.

Defence **The Random Solution: Address Space Layout Randomisation.** In 2001, Address Space Layout Randomisation (ASLR) was introduced, generally credited to the Linux Pax Project [176]. ASLR randomises the memory address of almost all parts of the application, but notably (1) the stack, (2) dynamically loaded libraries (like libc), and (3) the heap. This rather simple solution halts the return-into-libc attacks, as the starting address of libc needs to be known to find the location of a target function such as `system()`. Modern solutions randomises more application segments, notably the text (code) section.

Just like stack canaries, this technique breaks when the attacker can read out memory pointers or guess the value, determining the ASLR offset and the stack canary. Once the ASLR offset is known, any function address can be easily calculated[13]. The pointer memory read must not trigger a crash however, as a restart of the application will reset both the ASLR offset and the stack canary value. The simplicity and minimal overhead of both stack canaries and ASLR led to their widespread incorporation in every major operating system [62, 187, 47]. The ASLR paradigm has been extended for specialised use since, protecting specific smaller data structures. This has become known as *information hiding* within the virtual memory space [116, 23, 27] as discussed previously. Even in specialised use, the pointer must never be leaked and the memory area must be small enough not to be probed or guessed [78, 85].

One separate mechanism has been explored in defeating ASLR *without* leaking or guessing the offset. Göktas et al. explored a method of exploit writing where only

[13]See `https://libc.blukat.me/` for example.

relative addresses are used [86]. Relative addresses inherently contain the ASLR offset, so leaking the offset may not be necessary.

Attack **Generic Code-reuse: Return-Oriented Programming.** The concept of return-into-libc attacks was further generalised by Hovav Shacham in 2007 with the introduction of Return-oriented Programming (ROP) [166]. With ROP, more than one return address is (over)written on the stack. Instead of jumping to a libc function of choice, small code snippets are taken within the program and its libraries — ROP gadgets — that perform a useful task before returning. These are often a few instructions (e.g., a *pop* or a *mov*) at the end of various functions. By combining these, a shellcode can be written as ordered collection of these gadgets, called a ROP gadget chain. At the end of each gadget, the program will execute the `ret` and jump to the next ROP gadget. With enough ROP gadgets available, almost any payload can be written as long as the stack can be overwritten. This also saw a resurgence of shellcodes, as a ROP chain can grant a memory page write *and* execute permissions manually, enabling shellcode usage.

ROP also spiked a renewed interest in memory attacks, with people finding new ways of performing code reuse attacks. In 2010, Checkoway et al. suggested call-oriented programming where function pointers in indirect call instructions could be overwritten instead of return values [30]. One year later, jump-oriented programming also emerged as alternative to ROP [16]. Generating ROP chains can be done automatically by various tools [58, 160, 7, 204].

Defence **The Supposed End: Full Memory Safety.** Programming languages like Java and Python are considered memory safe as memory-based vulnerabilities do not appear in code written in these languages[14]. These languages either have memory-safe design or check memory bounds during run-time. Similar solutions have been suggested for C, including but not limited to HardBound in 2008 [64] and Soft-Bound in 2009 [136] to enforce *spatial* memory safety, and CETS in 2010 for *temporal* safety [137]. Other efforts focus on the integrity of data-*flow* within a program, notably Data Flow Integrity (DFI) [28] and Write Integrity Testing (WIT) [3] in 2006 and 2008 respectively.

Unfortunately, full memory safety and data-flow integrity are both prone to in-

[14]Less of bugs found in compilers/interpreters of such languages that may be written in a different language.

accuracies and a large overhead. For example, CETS reported a 116% overhead on applications to enforce full memory safety. This has been shown to be unreasonable in practice: after all, deciding to use C/C++ nowadays is often out of performance consideration. Hence, enforcing full memory safety in C/C++ is generally not done in practice.

Defence **No more jumping around: Control-Flow Integrity.** In a response to return-into-libc attacks and the generalised ROP attacks, Abadi et al. suggested Control Flow Integrity (CFI) [1]. CFI is two-fold: forward CFI and backwards CFI/shadow stack.

Forward CFI limits the allowed targets within an indirect `call` or `jmp` instruction. `call` instructions can initially be limited to the start of functions, functions who take the same amount of arguments (arity match) or even the same type arguments and return type (function type match) [188, 208]. Similarly, `jmp` targets can be limited to locations in its current function. Alternatively, static or dynamic analysis can determine expected behaviour for `call` or `jmp` instructions that can be used to whitelist allowed targets [149, 194].

Forward CFI ultimately needs to decide what are and are not allowed targets to `call` / `jump` to. More advanced techniques can limit the allowed targets even further, but can also break normal execution if the original application source code does not adhere to the same ruleset. Other techniques — namely dynamic instrumentation — can be more accurate without breaking regular execution, but add a large overhead [97]. Research distinguishes between course-grained CFI and fine-grained CFI, although it is unclear where the border lies exactly.

Backwards CFI requires checks on the return address. The accepted technique here is to use a shadow stack: a copy of the regular stack containing the return addresses, or hashes thereof [23, 59]. Similar to a stack canary, the value is checked against its original value and halts if the return address has been tampered with. Alternatively, the shadow stack value can be used as return address, assuming that the shadow stack cannot be tampered with. This specifically protects against ROP, although design and implementation details can easily break the mechanism [26, 60].

Control-flow integrity is not common in devices yet, but it is currently actively in development. Android kernels have CFI support for a couple of years at the time of writing, and the Linux 5.13 kernel has CFI capabilities too as of April

2021 [119]. In the meantime, Intel is working on support for hardware-enforced CFI on their processors called Intel CET, containing a shadow stack and indirect branch tracking. To use CET, Intel is developing a software component called FineIBT [120] for full "fine-grained" CFI.

At this point, we have arrived at the state-of-the-art in the real world, which is also where work by Szekeres et al. ends [182]. Everything following exists within research and is thus part of the NGME attack surface that we address in this dissertation.

Attack **Effortless: Automatic Exploit Generation.** Although writing exploits remains possible, it requires more and more effort over time, having to circumvent or defeat various defence techniques. As computers are commonly known to automate processes, so was the automation of writing exploits an interesting angle [21, 24]. The first work exploring the full breath of Automatic Exploit Generation (AEG) were Avgerinos et al. in 2011 [8, 29]. They utilise symbolic execution [39, 38] to trigger a stack-based buffer overflow and traverse its path constraints towards a compromised state.

AEG saw a spike of interest when DARPA organised the Cyber Grand Challenge in 2016 [81] where fully automatic machines participated in an attack/defence Capture the Flag tournament. The participating machines were programmed to patch security flaws on their services while attacking the other contenders, without any human interaction. This led to new tools [167] and techniques [178, 155, 168] in the space of a few years to automate the search, exploitation and patching of vulnerabilities. New techniques and tools mainly focus on one aspect within AEG, as the full breath of AEG consists of a large amount of smaller, distinct steps. One such aspect we focused on in the context of AEG is HLM, in Chapter 5.

Attack **Still Jumping: Control-Flow Bending.** The introduction of CFI means that attackers cannot jump anywhere in the code anymore, even if a memory leak is in place to defeat ASLR and stack canaries. However, an attacker can still choose which of the allowed `jmp/call` targets can be used. This can be used to perform Control-flow Bending (CFB) attacks where the `jmp/call` targets are chosen from allowed targets. It is shown that the range of allowed targets can be large enough to perform an attack [26, 60, 84, 43]. Even while using strict CFI policies, complex attacks are possible using CFB [25, 79]. These attacks abuse the *fundamental* limitations of CFI as a whole. Nonetheless, the attacks are relatively difficult and complex in practice. A small number of CFB attacks are currently known that defeat strict CFI policies.

Attack **No need to jump: Data-only Attacks.** With CFI becoming more and more refined, advanced ROP and CFB attacks are becoming more and more difficult. Thus, the offensive side needed a new tool. This gave birth to *non-control data attacks* or *data-only attacks*: an exploitation technique that does not override any instruction pointers or function pointers. Instead, it only overrides non-control data.

Data-only attacks are known since 2005, where Chen et al. showed real-world exploits without overriding any control data [32]. Yet, its popularity has been growing with the introduction of automatic tools to generate such exploits in 2015 [96]. In contrast to the direct data manipulation attacks, this explored the concept of modular exploit stitching by re-using existing code [34] using similar paradigms as ROP, called Data-Oriented Programming (DOP). Since then these attacks have been shown to be Turing-complete [98], generalised by Ispoglou et al. with (basic) Block Oriented Programming (BOP) [109] and with STEROIDS by Pewny et al. [151] in 2018 and 2019 respectively. Our focus in Chapter 6 is to prevent these types of attacks.

Defence **The resurgence: Data-flow Integrity.** As data-only attacks became popular in research, a defensive response is only a matter of time. This led researchers to revisit the concept of Data-Flow Integrity (DFI), blocking the core technique that data-only attacks rely upon [172, 35, 180, 139, 19].

To reduce runtime overhead, focus has been to limit instrumentation. For example, WIT [3] focuses merely on integrity (in contrast to confidentiality). Other DFI-based work is tailored towards certain user-defined object types [27], private class attributes [19], or a probing technique on whether data may be attacker-controlled [139]. Alternative efforts to limit overhead either require hardware support [172] or are tailored towards cyber-physical systems [180, 35].

The Conclusion: Has the cat caught the mouse? With every defence, a new attack technique quickly becomes the norm. With every attack again, new defences are presented and implemented — especially if the overhead of said new technique is acceptable (or can be made acceptable). This raises the question whether we are merely at the next step in a game of cat and mouse.

Until we achieve complete memory safety within a reasonable overhead, some attacks will always be possible. Attacks are however becoming increasingly complex and context-dependent: specific tools need to be used (e.g., data-only gadgets) that are not transferable to a different application. Furthermore, the payload of such attacks may not be as clear, since payloads too become application- and context-dependent. Instead, attackers may e.g., leak a private key or overwrite an "authenticated" boolean [32, 51, 98]: data of which the location and method of accessing varies across different applications.

Note that data-flow integrity is limited w.r.t. analysis accuracies in the same way that CFI is. This means DFI raises the bar for exploit writers, but DFI does *not* guarantee the absence of data-only attacks. It is not unlikely that new attacks and attack techniques will be found on the residual attack surface. Then, the game starts over again.

2.6 Crowd-sourcing Research, Awareness and Education

Crowd-sourcing has been used in several scientific domains to solve a range of research tasks. Researchers have sought to motivate their crowd through financial gains [168]; the idea of being part of a research breakthrough [117, 72]; or by entertaining them through a fun game [199, 200, 44], i.e., *gamification*. Where the former two are often repetitive and non-challenging tasks, gamification *relies* on the task at hand being challenging: something to e.g., replace a sudoku with. These are branded as scientific discovery games, where the goal is to engage the general public in a series of puzzles that hold scientific significance.

Arguably the most successful scientific discovery game is FoldIt [44, 115, 83, 114], a protein-folding game that made both awareness campaigns and scientific discoveries in the aid of AIDS, Cancer and Alzheimers, to mention a few. This game is essentially an optimisation problem on the folding of proteins. The FoldIt team has performed extensive research on how to build such a game [45, 5] with over 850,000 players. They also identified key factors for building a successful game [45] — that inspired our design in Chapter 5.

With regards to a game's **design**, it must *reflect and illuminate the natural rules of the system*. After all, the game is meant to be played by players who have little to no experience with protein folding or biology in general. This means that the important parts of the game system need to catch the eye of the player through its design. Within FoldIt, this is done using additional objects (e.g., showing bubbles where the void space needs to be filled up) and colours (e.g., using red upon illegal clashes). Furthermore, they use a gradient between red and green on different parts of the protein to show where the most points can be gained (and improve upon the optimisation problem). At the same time, it must *manage and hide the complexity of the system*. As a clear example in the FoldIt game, hydrogen atoms are hidden by default as they exist in very large quantities and do not add structural information. If hidden parts become important — for example when they cause a clash — they will reappear if necessary for the understanding of the player. Thus, managing this complexity is vital to the understanding of the player. The game also needs to be *approachable*, as is done in FoldIt through a bright and cartoonish look.

While **playing** the game, everything a player does must *respect the constraints of the system* that is being explored. This means the result must be a solution to the real-world problem. Hence, the game should incorporate known algorithms and optimisation techniques, as is described in detail in their work. At the same time, the players need to have enough meaningful interactions: *sufficient to explore* the problem-space and come up with new solutions. To make the game more *intuitive and fun*, the FoldIt game aims for a direct interaction model with the protein, as if you could actually touch it. Furthermore, the game has a *scoring system* directing players to the solution, even though solutions to real-world puzzles are not known.

Gamification has been used in the field of information security before, mainly focusing on educational value. For example, board games have been developed for computer security education [89, 88]. To the best of our knowledge, we are the first to deploy gamification within information security as crowdsourcing technique. For our crowdsourcing-based project, see Chapter 5.

2.6.1 Education

Similar to FoldIt, our approach in Chapter 5 works on the principle of learning by doing [76, 154, 36]. On the other hand, whilst FoldIt introduces players to real-world scenarios immediately, we start with a series of tutorials in order to train the public. This way, we follow a pattern of "teach $\rightarrow$ crowdsource". In order to retain educational value, we attempted to minimise inaccuracies on the visual elements and no abstraction inaccuracies that can affect the real-world value in any way. This way, we ensure that the skill from playing the game remains transferable to the effective HLM solving skill. These inaccuracies are discussed in Chapter 5.

Within the teaching phase, new concepts are taught through an structured learning assessment feedback loop [42]. Each level-up phase consists of a brief explanation of a new topic, followed by a dummy assessment to test the new knowledge. In contrast to FoldIt we do introduce explicit tutorials, but we follow the same mindset of FoldIt [5] by only explaining the core concepts that cannot be easily learnt by playing directly (e.g., complex heap manager fitting methods). Even with this new knowledge, it is mainly mastered by playing [36] the puzzles that follow, *after* the level-up challenge. It is here that players run into a vast set of variety within puzzles where combinations of new concepts prove challenging. Once the player is confident enough, they can decide to move onto the next level-up phase which adds the next challenge. This is necessary since we observed that players tend to lose interest when the game moves either too quick or too slow. Players' interest and motivation however is important now only for retaining the player, but also has a strong effect on the quality of gamification [91]. Once all core concepts have been explained, the game moves onto the crowdsourcing aspect, where the answers found by players can be useful in a scientific context. On top, our game in Chapter 5 itself hopefully raises awareness on the dangers of (memory) vulnerabilities as an overarching result, which will be discussed later.

Open your eyes and then open your eyes again.

TERRY PRATCHETT

3 The Exploitation Process

Exploiting an application is a complex and multifaceted task. This dissertation follows a three step paradigm to break down the exploitation process as shown in Figure 3.1. It all starts with finding a *vulnerability*. Here, we question what a vulnerability truly represents. We also need to know what the vulnerability can affect; and if and how the vulnerability can be reached. The vulnerability needs to be *found* and *understood*. These questions are explored in depth in Chapter 4.

This vulnerability then needs to be leveraged to accomplish a certain goal, e.g., gaining control over the underlying machine. We need to reason in a forwards manner to see what we can accomplish with the vulnerability, before we can formulate a goal that may be possible. In other words, we need to figure out what level of *control* we can gain from the vulnerability. In part, this step is comprised of a list of techniques. We attempt to get the most out of the vulnerability, regardless of the goal of the exploit. For example, Return-Oriented Programming (ROP) is a control technique: it does not specify any payload but provides the attacker with arbitrary code execution, enhancing control. In order to perform ROP, one needs a large enough write primitive to write the return pointers and data on the right location(s) of the stack. This may require more control techniques before ROP can be performed. How to gain control for heap-based NGMEs is the focus of Chapter 5.

Finally, the attacker can use the control gained to perform the compromising task: executing a *payload*. Although separate, the aim of the attack *guides* the control step

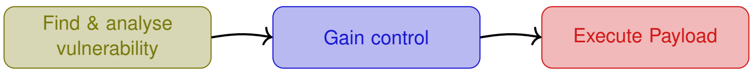


Figure 3.1: The exploitation process in three rough steps.

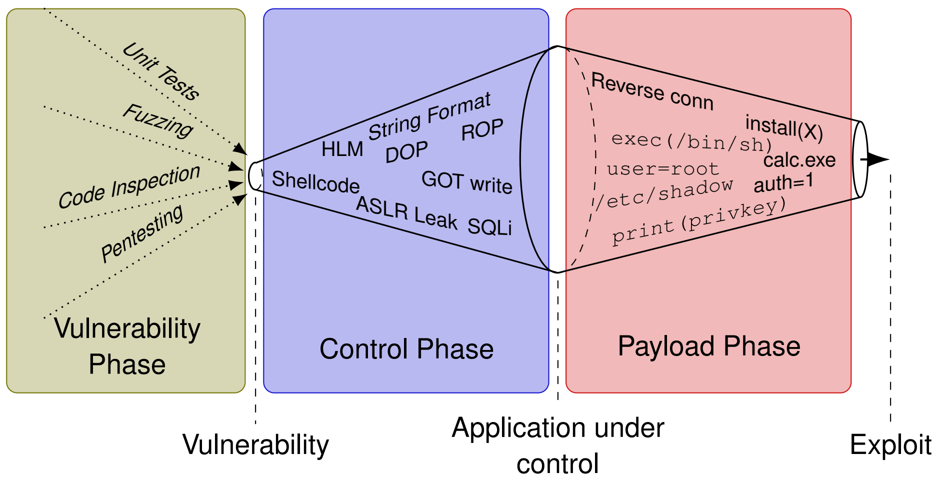


Figure 3.2: Here we depict the 3-step exploit writing process. Many angles in the vulnerability phase (yellow-green) can lead to a small, initial point of entry into unintended behaviour of the application. Afterwards, we use control techniques (blue) to broaden our control. In the payload phase, we narrow this back down towards our wanted behaviour to complete the exploit.

into what control is required for the exploit. Depending on the results of the control step, the exploit writer may need to alter the payload to fit the level of control gained. While the control step broadens the search for capabilities, the payload phase narrows it down towards a concrete goal. Preventing payload execution for NGMEs is the focus of Chapter 6. The three steps are visualised in Figure 3.2.

The procedure can be compared to a mathematical proof. Certain properties are known (i.e., the application purpose and (byte)code, possibly one or more vulnerabilities), as well as the end result: what we want to do with the final exploit. How to link the two is the challenge, and often requires both forwards and backwards reasoning to find out if such a proof exists (a proof of exploitability in our case), and how this proof is constructed. The three steps as outlined above provide a guideline on how to form this proof of exploitability.

Exploitation is sometimes regarded in a larger scheme with additional steps, such as reconnaissance [99], persistence [74] or remaining undetected [61]. Although these are undoubtedly important steps in a real-world scenario, we are only concerned with the core vulnerability exploitation aspect of the process. Furthermore, unlike our three step exploitation model, such schemes typically view the exploitation (development) process as a single phase.

```c
// Remains in memory if keep-alive is set
// Target structure
typedef struct {
  char *ty;
  int socket;
  [..]
} connection_t;

// Bugged structure
typedef struct {
  [..]
  char *dir;
  size_t content_length;
  char *body;
} request_t;

// CGI base directory global
char *exec_dir;
```

Listing 3.1: Example vulnerable C code *header* as explained in Section 3.1

In the remainder of this chapter, we will discuss the three step exploitation process in more detail. In particular, we will systematise the process where possible through analysing relevant primitives and techniques, discussing in each step what is required for each technique and how to systematically defend against certain techniques. This provides a framework for the following chapters where we will discuss primitives, a control technique and a defence technique in chapters 4, 5 and 6 respectively.

3.1 An Example: The HTTP Server

To provide an intuitive understanding of the paradigm introduced in this chapter, we present a fictitious vulnerable HTTP server. This HTTP server is a single-threaded, single-processed application without ASLR, running under port 80 and available to the attacker. The relevant C code is shown in Listing 3.2 (placed at the end of this chapter due to size) and a header is shown in Listing 3.1. The HTTP server will be a running example in the remainder of this dissertation.

The vulnerability is a simplified adaptation of CVE-2002-1496 [49], originally a heap-based buffer overflow in a webserver called nullhttpd. This CVE first came to the attention of the research community by Chen et al. [32] as a vulnerability that could be abused *without* overwriting any control pointers: a data-only attack.

In our code snippet in Listing 3.2, we show the use of a similar vulnerability. We will go into detail on exploiting this vulnerability later in this chapter, but now we briefly cover each relevant aspect of this code.

Following our three-step paradigm, we first need to find a vulnerability. This can be done through e.g., a manual code review, fuzzing or symbolic execution, as will be discussed in Chapter 4. In this case, we can see that the (user-controlled) `content_length` is used on line 31 to create a heap buffer, after adding 80 more bytes. This is not checked against a potential integer overflow, so the buffer size could end up smaller than the original content (where `content_length`+80<`content_length`). Ultimately, this leads to a heap overflow from this buffer into adjacent heap memory, on line 33.

To gain more *control* with this vulnerability, we may be able to overwrite the `connection_t` structure, which contains a pointer to the content-type string. The string value is under control of the user as seen on line 25. By overwriting the pointer-value, we can choose exactly *where* the content-type is written. We also can already decide *what* is being written, meaning we can decide what to write where.

As a *payload*, we want to execute an application. We can do this through the `exec` function on line 5. Yet, any slashes are removed in our application (line 6) so it can only execute applications from the allowed directory set on line 3. In other words, our goal is to overwrite the *exec_dir* buffer value. We can change it into the directory where our desired application resides (e.g., `/bin/sh`), with user-chosen arguments. This provides us with a shell on the webserver: our exploit is complete.

3.2 The Vulnerability Phase

In order to exploit an application, something needs to be wrong. As simple as this sounds, the concept of a vulnerability is debated to this day. We will discuss this in detail in Chapter 4, but it is important to understand that a vulnerability is a weakness in a computer system that could affect the behaviour of the system in unintended ways. Concretely, we can represent the vulnerability as some input that does not provide the expected output.

Representing the vulnerability by its input is tied to the reachability problem of vulnerabilities. In our HTTP server example in Section 3.1, we discussed a vulnerability we could see from the source code. If the vulnerability cannot be reached (e.g., because of additional checks or assertions in other functions), there is no reason to worry. The reachability problem is equivalent to the halting problem [191] and hence undecidable[15]. In our example, a request with a set content length of at least `SIZE_T_MAX-79` will get to trigger the vulnerability as far as is shown within the code snippet.

While the focus of this dissertation is on memory violations, we extrapolate here and in Chapter 4 onto any vulnerability. Chapter 4 discusses a taxonomy across *all* vulnerabilities in a more generic setting. In this chapter, we mainly focus on the fundamental difference between exploiting a memory vulnerability and a different type of vulnerability.

3.2.1 The Vulnerability Primitive

The vulnerability primitive is three-fold. First, we need one or more (direct or indirect) inputs that trigger the vulnerability, ensuring it is reachable and knowing how to get there. Secondly, we need to find the right perspective on the abstraction level where the vulnerability bears meaning. As discussed in the weird machine model limitations (Section 2.1.1), an SQL injection has a completely different model compared to e.g., the buffer overflow example from Section 3.1. Lastly, we need to understand the implications of the vulnerability, what violation caused the existence of the vulnerability and thus can be used to write an exploit.

For example, a buffer overflow exposes some form of memory corruption (violation) through undefined behaviour when compiling (abstraction level) but needs to be reachable with some input (trigger). Afterwards, we need to know what size overflow occurs, where and when it occurs, what characters are allowed, et cetera. In contrast, a reachable SQL injection vulnerability exposes user-data that will be incorrectly interpreted as SQL code. For an SQL injection vulnerability we also need to know its implications and limitations, for example knowing what characters are allowed and knowing what query we are changing. Note that they

[15]This is done easily by proof of reduction, by adding transitions from every halting state in a Turing machine into one single state without any outgoing transitions. If we have a machine that can decide whether to reach that state, we have built a halting machine.

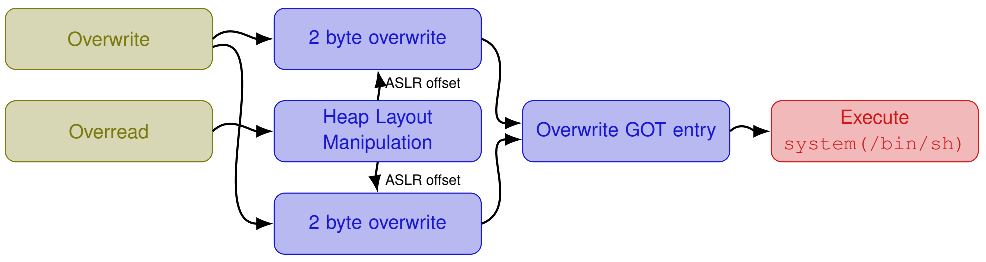


Figure 3.3: This figure shows a model exploitation chain with 2 vulnerabilities. A thorough explanation is available in Section 3.3.1. The vulnerability steps are depicted in yellow-green, control steps in blue and the payload steps in red.

act on a completely different abstraction layer, which changes the applicable control techniques and payloads. This is the topic of Chapter 4.

3.2.2 Vulnerability Defences

A natural way to protect against vulnerabilities is to locate and patch the mistakes, but this has proved insufficient in practice. In addition, fundamental techniques can be deployed to stop the expression of vulnerability classes entirely. One example is enforcement of memory safety as discussed in Section 2.2. Even though C is a memory unsafe language, full memory safety can be achieved through the use of Hardbound [64] or Softbound [136] in combination with CETS [137]. It must be noted that these solutions present a large overhead and do not see any use in practice. Instead, automatic memory management (e.g., garbage collection, runtime bounds checks) is used with other programming languages like Java or Python.

3.3 The Control Phase

Once a reachable vulnerability is established, an initial level of control over the application has already been achieved. The goal in step 2 is to enhance this level of control, ultimately to execute step 3 (the payload).

Many techniques exist to move from one level of control to another. For example, an off-by-one string vulnerability could be placed next to a second string that overwrites the null byte of the first string to create a larger contiguous buffer overwrite. If we use this to overwrite a user-data pointer, it can become an Arbitrary Write Primitive (AWP), and ROP can turn an AWP into Arbitrary Code Execution (ACE). The three steps above are all individual and independent techniques, but in the right circumstances they present a way to turn an off-by-one vulnerability into arbitrary code execution.

All of the above are meaningless with a different vulnerability primitive. In a poorly sanitised SQL pipe, the initial level of control is completely different, and SQL injection techniques are available to enhance this control. The control step starts with the initial level of control as gained through the vulnerability or vulnerabilities. In short, the control step is the "glue" of the exploit, linking the entry point(s) to the goal.

3.3.1 Initial Control

The vulnerability primitive presents an initial level of control. This considers the full context of the vulnerability and thus is highly specific. For example, Heartbleed/CVE-2014-0160 [51] is a heap buffer overread vulnerability where the size of the overread is user-controlled. Control over what is being overread is limited by the maximum amount of bytes to read, the heap state being unknown and attacker heap interaction itself being limited. On the other hand, Shellshock/CVE-2014-6271 [52] presents a passthrough into the bash shell due to poor sanitisation. What can be executed here is completely dependant on the applications available on the device (and allowed by access control). Furthermore, it is limited to a certain set and length of input characters mishandled by Shellshock, and dependent on the underlying machine, software and environment.

The initial control as exposed by a vulnerability differs per vulnerability and per environment. This needs to be taken into account when starting the control phase. First, we need to consider any factor that affects the behaviour of the vulnerability. Second, we determine which of those are user-controlled and/or malleable by the exploit writer. Third and most importantly, what *exactly* is exposed by the vulnerability: where do we start in enhancing control. Unfortunately, this is very specific to a vulnerability, and generalising this is out of scope for this dissertation. Szekeres et al. have attempted this specifically in the context of memory vulnerabilities,

Abbrev.	Name	Effect	Comment
CWP	Contiguous Write Primitive	Write N chosen bytes on an unchosen location	
AWP	Arbitrary Write Primitive	Write N chosen bytes on a location of choice	Also known as Write-what-where
ARP	Arbitrary Read Primitive	Read N chosen bytes on a location of choice	
TC	Turing-Completeness	Perform any calculation of choice	May not be able to execute system calls/interrupts
ACE	Arbitrary Code Execution	Execute any series of instructions	Generally constitutes full compromise of the application

Table 3.1: This table lists some generic control levels with their meaning and effect.

together with a simplified flow of gaining control, as shown in their SoK [182], Figure 1.

Combining Vulnerabilities. More than one vulnerability can be used to gain a certain level of control. This means we could have more than one point of initial control in the total exploit, glued together with control steps. For example, if we have an overread vulnerability paired with an overwrite vulnerability, we can use the former to break ASLR whilst using its results to overwrite into a meaningful pointer (relative to the ASLR offset). Together, this written pointer can be used to e.g., overwrite or overread other values. This is visualised in Figure 3.3, where the blue blocks represent control techniques used.

3.3.2 Control Primitives

From the initial control of the vulnerability, we need to build a level of control that is suitable to execute a given payload. This is often a series of individual steps where each step requires a level of control and provides an elevated level of control. Some generic control levels are listed in Table 3.1.

The visual example in Figure 3.3 contains two vulnerabilities. The first vulnerability is an overwrite vulnerability where we control *what* we write and *where* we write, but we are limited to 2 bytes (e.g., through a complex format string vulnerability). The second vulnerability is a heap overread vulnerability. We use Heap Layout Manipulation (HLM, as will be discussed in detail in Chapter 5) to make sure we read into a pointer value. From this pointer value, we can determine the ASLR offset that we can use in the overwrite vulnerability. We can perform the

Name	Requires	Provides	Comment
ROP	AWP	ACE	No shadow stack
DOP	AWP	TC	Given enough gadgets and dispatcher
HLM	Heap violation	AWP / Control Hijack	
Unsafe unlink	Heap violation	AWP / Control Hijack	No metadata protection
SQL Injection	SQL sanitisation bug	db read/write	
Blind SQL Injection	SQL sanitisation bug	db read/write	No output
GOT/vTable overwrite	AWP	Control Hijack	No forwards CFI
Shellcode	Control Hijack	ACE	No W⊕X
ROP W⊕X bypass	AWP	Shellcode enabled	W⊕X
Custom dispatcher	stack overwrite at the end of function	action repeatability	overwrite the saved IP and BP
netcat reverse connect	`execve` execution	remote shell	can be considered payload
String format "%n" attack	string format bug	AWP	
Overwrite2Overread	AWP	ARP	Overwrite printed pointer
ASLR Offset leak	ARP	ASLR Leak	
Stack Canary leak	ARP	canary leak	

Table 3.2: This table lists some control primitives, mainly in the existence of a memory violation. (ROP = Return-Oriented Programming, AWP = Arbitrary Write Primitive, ACE = Arbitrary Code Execution, TC = Turing-Completeness, HLM = Heap Layout Manipulation, ARP = Arbitrary Read Primitive)

overwrite twice[16] to overwrite a Global Offset Table (GOT) entry into the location of the `system` function, using the ASLR offset in both the GOT location and the `system` location. To execute the payload, we call whatever GOT entry has been overwritten with `/bin/sh` as the first argument to spawn a shell.

Generic Control Primitives. Some control primitives are generic enough to work under most circumstances. One such generic technique called ROP provides an ACE primitive, once we can overwrite enough bytes at the right location. Using the example from Figure 3.3, ROP can technically be used with the arbitrary overwrite vulnerability and broken ASLR (instead of overwriting the GOT table). This would however require a very consistent dispatcher that repeats the 2-byte overwrite on the stack many times before starting the ROP chain execution. This would be tedious but potentially necessary: overwriting the GOT table entry is much simpler, but cannot be used if e.g., the application is statically linked.

This can again be overcome if a system call to `mmap/mprotect` can be performed with a (shorter) ROP chain, to create a writable and executable memory segment. Afterwards, we need to write a shellcode there and overwrite one last

[16]Here we assume that the difference between the original pointer and the `system` function pointer differs my 4 bytes maximum. Alternatively, we can repeat the process to overwrite more bytes.

saved instruction pointer to jump to our shellcode. If possible, this technique limits the size of the ROP chain, and re-enables shellcode techniques.

Control techniques do *not* guarantee success but do provide the exploit writer with a useful toolset. All these techniques work in the right circumstances when requirements are met. A variety of generic control primitives is listed in Table 3.2.

Note that the control primitives in Table 3.2 are simplified. For example, ROP requires a fairly large write primitive, something that is not listed. On the other hand, ROP can be performed with a large enough contiguous stack overflow under the right circumstances[17]. This does not discredit the control primitive but requires caution.

Some modern techniques are proven to be Turing-Complete (TC). This is a *limited* form of ACE, because TC in itself provides no control over input or output (i.e., through system calls, see Section 2.4). Whilst any calculation is possible from a mathematical standpoint, the results are impactful when combined with input or output. In the (un)fortunate event where no system calls are available when gaining TC, TC is completely harmless — in contrast to ACE.

3.3.3 Control Defences

In Chapter 2, we discussed the most important developments in memory exploitation, including defence techniques. Unsurprisingly, most defence techniques for binaries are tailored towards preventing control techniques. For example, a non-executable stack was implemented to prevent the usage of shellcodes on the stack. Similarly, backwards CFI prevents ret2libc or ROP attacks and forwards CFI protects against control-hijacks through indirect calls and jumps, e.g., overwriting GOT records or vtables.

3.4 The Payload Phase

While the control step broadens the scope, the payload narrows it back down. If the level of control does not support the execution of the payload, we either need

[17]Among others, this requires the absence or leaking of stack canaries and potentially requiring the capability of writing null bytes.

to revisit the control step; reconsider our the payload option(s); or accept defeat. The payload is a *concrete* method of accomplishing the *goal* for the exploit.

We can often determine a number of payloads that may be suitable. For example, a goal like "steal a private key" is not concrete and hence cannot be linked to control techniques. Instead, we can determine *how* to steal the private key, which could be done by (1) reading the right area of the application: e.g.„ through Arbitrary Read Primitive (ARP), (2) write the private key into a printable area (e.g., through an AWP), (3) gaining a shell to read out the private key, and so forth. At that point, we can link our payload to explicit levels of control. Each individual payload option, with its required control primitives, is a *payload primitive*.

Generic Payloads While some payloads gain meaning through the application under attack, other payloads are transferable to other applications. As an example, shellcodes can be reused since they perform a generic operation after gaining ACE. Generic payloads typically target the underlying machine (e.g., gaining a shell or installing a backdoor), since the semantics of the application under attack is irrelevant. Similar to generic primitives, generic payloads can be chosen by the exploit developer when the concrete payload suits the overall goal of the developer. A more in-depth discussion on generic payloads is presented in Chapter 6.

3.4.1 Payload Defences

For a payload level defence, we assume the control phase has been successful by the attacker. The application at this point is compromised, so we have very little guarantees towards protecting the application itself. Payload level defence techniques often have a broader view than just the application internals. Instead, access control or virtualisation limits the impact in the case of complete application compromise.

With defence techniques in place, more guarantees can be provided even in the case of compromise. These guarantees can be used to provide a payload-step defence technique, as is the main topic of Chapter 6.

3.5 Next Generation Memory Exploits

In Next Generation Memory Exploits — where a fine-grained CFI is deployed alongside current defences — multiple generic control techniques are no longer applicable. CFI blocks most targets in returns and indirect calls and jumps, notably breaking the ROP control technique. It does not provide a unique target however, providing at least some room for exploit writers to decide on the target. In some circumstances this can lead to successful exploitation [79]. One generic control primitive has been identified in the presence of two consecutive calls to functions of the `printf` family [25] given some level of prior control. The resulting control provides exploit-writers with TC.

Another angle is to forget about control-data completely [32]. Instead of breaking CFI, it is circumvented by not changing any control-data. Non-control data can be changed and moved around enough to gain TC [98], or to execute payloads under the right circumstances [109, 156, 151]. Early examples overwrite meaningful data to bypass authentication or execute a different binary than intended: this is referred to as Direct Data Manipulation (DDM). More recent attacks perform code-reuse attacks and pointer stitching to change the overall semantic meaning of the application under attack, called Data-Oriented Programming (DOP) [34]. DOP attacks require a large code-base and are almost exclusively generated automatically.

The CFB and Data-only attack vectors can be seen as fundamentally different: CFB explicitly leverages the inaccuracies of CFI whilst Data-only Attacks strictly adhere to the control-flow. It raises the question why exploit-writers would stick to one technique over the other, or more importantly: over a combination. If the CFI target set cannot be limited further[18], both approaches would be equally suitable and usable once CFI is widely deployed. Transferring control into a target of choice can be hugely beneficial to find DOP gadgets or dispatchers. Current DOP techniques are very computationally expensive and complex, so adding CFB techniques adds to the modelling efforts and computational limitations[19].

The interchangeable nature of CFB and Data-only Attacks can be seen clearly with UCT [97]. UCT is a CFI solution that rewrites any indirect control-flow trans-

[18]Without incurring unreasonable overhead.

[19]The computation limitations are currently mainly state explosion issues in symbolic execution and effective guiding in fuzzing.

fer into a switch-like structure[20]. Based on the original heuristics, it decides what control-flow transfer will take place, and performs a direct jump to the respective direct control transfer. The authors state that any control-flow deviation is now a non-control data issue, which is out of scope for their work. In other words, the authors have successfully translated the CFB problem space into a Data-only problem space. Thus, attempting a data-only attack on an application hardened with UCT could provide better results.

We believe the classification and labelling of various control techniques is an important and useful tool for exploit writers, developers and researchers alike. However, it should be taken into account that hybrid NGME attacks are possible too. If DDM is good enough to gain the necessary level of control, there is no need to search for DOP dispatchers. On the other hand, a CFB technique can become fruitful in creating a DOP dispatcher when no natural dispatchers can be found or used.

3.6 The HTTP Server Attack

Vulnerability. In Section 3.1, we discussed a fictitious HTTP server (Listing 3.2) with a vulnerability on line 31. We can now apply the three phase exploit development process onto this HTTP server to build an NGME. To recap, we call `malloc` on "`content_length` + 80", to cause an unsigned integer overflow. An unsigned integer overflow is not an issue by default as it is properly defined in C. In this instance however, it is undesirable and its resulting behaviour will differ from the required behaviour. In our case, it creates a heap overflow vulnerability on line 33, as the buffer is smaller than the content-length specified. It is however at the integer overflow where the vulnerability appears, because this is where behaviour diverges. This is the topic for Chapter 4. This initial vulnerability opens up a path in the code into the heap buffer overflow.

Control. Given the heap overflow, we can perform heap layout manipulation to enforce the overwrite of a data pointer, for example the `content_type` pointer. This can then be used as pointer to write a string-value on a chosen location on

[20]Do note that UCT is paired with a run-time monitor to further narrow allowed the (originally indirect) targets.

Line 25 in Listing 3.2. This creates an AWP. The AWP is limited: we write a string-type value, so our write must end with a null byte and will not contain additional null bytes. Considering it is all part of an HTTP request, additional characters may need to be escaped before use. Overflow size is no limitation in the given AWP. How to perform heap layout manipulation in a simple, heap agnostic way is discussed in Chapter 5.

Payload. On Line 3 a Common Gateway Interface (CGI) base directory is set. This directory is preprended on Line 8 before the `execv` function is called, originally to ensure the list of applications is limited. Overwriting this value opens up the functionality to any application of the device, through a DDM attack. For example, writing `/bin/` as CGI base directory would enable a call to `/bin/sh`. A diagram of the full exploit is shown in Figure 3.4.

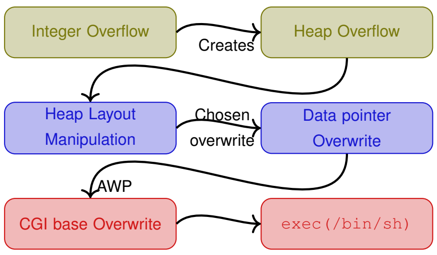


Figure 3.4: This figure shows the vulnerability/control/payload steps for an example exploit in our example from Section 3.1. The vulnerability steps are depicted in yellow-green, control steps in blue and the payload steps in red.

Calling `/bin/sh` on a remote webserver does not help in gaining control, unless we can interact with the resulting shell. More realistically, one would set up a reverse connection shell, e.g., with netcat[21]:

$$/usr/bin/nc\ [attacker\ public\ ip\ address]\ [port]\ -e\ /bin/sh$$

To complete the exploit with our reverse shell, we must set the CGI base directory to `/usr/bin/` and then call netcat with the above arguments. This does truly present us with a full-control data-only exploit. How to halt this attack without changing functionality, system-wide modifications or manual intervention is the topic of Chapter 6.

[21]This only works on older versions of netcat, as the developers concluded it might be a security risk to add a direct option for connecting to a remote machine and piping its communication straight to an application of choice.

```c
void init() {
  exec_dir = malloc(128);
  strlcpy(exec_dir, "/usr/bin/cgi/", 127);
}
uint8_t exec(char *application, char **argv) {
  application = escape_and_remove_slashes(application);
  char *full_path = malloc(128);
  strlcpy(full_path, exec_dir,127);
  strlcat(full_path, application, 127);
  int fork_res = fork();
  if (fork_res==0) {
    execv(full_path, argv);
  } else if (fork_res<0) {
    perror("execv");
  } else {
    free(argv);
    return EXIT_SUCCESS;
  }
}
request_t *parse_request(connection_t *conn, char *request_string) {
  request_t *request = malloc(sizeof(request_t));
  char *content_type = parse_content_type(request_string);
  if (conn->ty==NULL || strcmp(content_type,conn->ty)) {
    conn->ty = realloc(conn->ty, strlen(content_type)+1);
    strlcpy(conn->ty, content_type, strlen(content_type));
  }
  [..]
  request->dir = parse_directory(request_string);
  request->content_length = parse_content_length(request_string);
  char *body_start = find_body_start(request_string);
  request->body = malloc(request->content_length+80);

  strlcpy(request->body, body_start, request->content_length);
}
void handle_request(connection_t *conn, char *request_string) {
  request_t *request = parse_request(conn, request_string);
  if (strncmp(request->dir, "/cgi-bin/",9)===0) {
    exec(request->dir+9, parse_cgi_args(request->body));
  }
  [..]
}
```

Listing 3.2: Example vulnerable C code as explained in Section 3.1

Manager: You have to fix all the bugs in the C++ compiler, but you can't change the behavior in any way.
Me: That's not possible. By definition, if you fix a bug, the behavior is necessarily different.
Manager: Brian, you don't understand. You have to fix the bugs but the compiler's behavior can't change.

Brian W. Kernighan, in UNIX: A History and a Memoir

The Vulnerability: GENerically understanding them

4

As discussed in Chapter 3, an attacker requires at least one vulnerability to write an exploit. Defining the concept of a vulnerability may seem easy at first sight, but a universally agreed upon definition does not exist. Moreover, some definitions are too narrow or too broad, which has led to the introduction and usage of separate terminology, such as bugs; weaknesses; flaws; mistakes; errors; and so on. This too is done on a per-case basis, so its usage is inconsistent.

Intuitively, when a vulnerability exists, the application behaves different compared to the expected behaviour. In addition, a vulnerability assumes something about the abusability of this change in behaviour. We believe the change in behaviour is key to understanding vulnerabilities.

Once we understand the nature of vulnerabilities, we gain a number of advantages. For example, we know what control techniques apply to what vulnerability, leading us to the next step in exploit writing. We also learn how and where to patch the vulnerability, empowering us to protect ourselves. On top, we can determine what vulnerability types can be used to write NGMEs while rendering other vulnerabilities irrelevant.

Concretely, this chapter discusses the limitations of existing definitions of bug and vulnerability. Then we define new, fundamental properties of bugs and vulnerabilities to form a new taxonomy and definition. We show these properties are naturally embedded in bug-finding techniques and propose a new generic methodology for bug-finding. Afterwards, we discuss how these properties can aid towards vulnerability reduction and how classifying vulnerabilities can help to reason about their potential consequences. Then we highlight NGMEs, discussing what subset of vulnerabilities are relevant to the NGME attack surface before concluding.

Definition by	Exploitability	Policy	Assets	Threat	CIA
FIPS 200 [140], NIST SP 800-128, CNSS	✓*	×	×	✓	×
NIST CVE [130]	✓	×	×	×	✓
MITRE CWE [131]	×	×	×	✓	×
RFC 4949 [100]	✓*	✓	×	×	×
ISO 27000 [106]	✓	×	✓	✓	×
ISO 27005 [107]	✓	×	✓	✓	×

Table 4.1: Comparison of various definitions of "vulnerability", by whether it mentions the following: exploitability; violating the security policy; affecting assets; to be used by a threat source; and the Confidentiality-Integrity-Availability triad. *: Mention the potential exploitability rather than exploitability.

4.1 Existing Vulnerability Definitions

Different authorities use different definitions for bugs and vulnerabilities. Some definitions even vary within the same authority. Table 4.1 shows a breakdown of vulnerability definitions by various authorities.

All definitions refer to a vulnerability as a weakness, most of which mention exploitability[22]. The effect a vulnerability has is generally divided into three categories: whether it affects the companies assets; whether it can violate the security policy; and whether it can affect the CIA properties. Furthermore, some definitions explicitly refer to usage by a threat actor whereas others do not.

All properties ultimately question what is and what is not a vulnerability. For example, the exploitability of a vulnerability is an undecidable problem [70] and can hence only be proven by providing an actual exploit. Then, the impact of the given exploit may or may not impact the security policy or a specified asset, which again questions whether it is a vulnerability — depending on the definition used.

Practitioners who work with vulnerabilities on a daily basis know what is and what is not a vulnerability. Yet, no *technical* definition exists that describes what a vulnerability is.

[22]Some definitions mention the *potential* exploitability instead. In Table 4.1 this is marked with an asterisk(*).

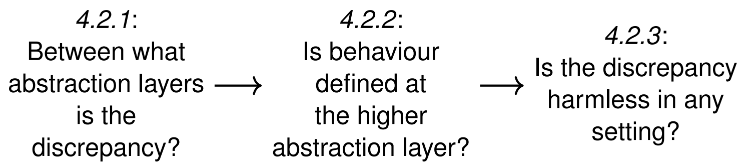

4.2.1: Between what abstraction layers is the discrepancy?
4.2.2: Is behaviour defined at the higher abstraction layer?
4.2.3: Is the discrepancy harmless in any setting?

Figure 4.1: Bug/Vulnerability assessment steps.

4.2 GENerically understanding Vulnerabilities in applications

As we develop software applications, we form specifications from a set of requirements. If the specification of an application (or lack thereof) differs from its actual implementation or execution, we refer to this discrepancy as a *bug*. Our taxonomy — GEN — concretises this. With GEN, we view the specification of an application and the execution as different abstraction layers, and consider discrepancies *between* such abstraction layers as the origin of bugs. In addition to *where*, GEN considers *why* a bug occurs: through the *incorrectness-undefinedness property*.

Finally, although these two properties serve to classify a bug in GEN, they do not say whether the bug is in fact a *vulnerability*. In GEN, we define a *vulnerability* as any bug that potentially exposes new data or functionality (e.g., privilege escalation), as will be discussed in Section 4.2.3.

The GEN conceptual model thus results in a three step vulnerability assessment process, as illustrated in Figure 4.1. We next discuss each of these steps in detail.

4.2.1 Abstraction Layers

In GEN, we consider a number of coarse-grained abstraction layers, as illustrated in Figure 4.2. A bug or vulnerability in this notion is a discrepancy between two layers. This occurs when two layers contradict: one layer specified to execute operation A whilst the other layer executes A'. The contradiction *between* the two layers describe the vulnerability. The granularity of the abstraction layers in GEN are carefully designed and forms the basis for a generic solution. A further discussion on custom layers can be found in Section 4.8.

We will first discuss the layers A—G as depicted in Figure 4.2, describing the steps from the most abstract representation to a hardware "abstraction" layer, where the application concretely runs on a piece of hardware. Afterwards, we will discuss the two top-level layers in Figure 4.2, the security policy and human-computer interaction. These layers differ considerably from the others: they do not describe the behaviour of the actual application, but rather define requirements and interactions respectively.

A **Informal Description.** First, an application has an **Informal Description**, describing what the application is supposed to do. An informal description can take on many forms, from a verbal one-liner to a comprehensive specification document (e.g., an RFC). Note that in addition to the development of an application, the informal description can also describe the deployment of an existing piece of software in a given setting.

B **Formal Specification.** Second, we may have a Formal Specification of the behaviour of the application. Here we refer to a mathematical model where properties can be proven (e.g., through model checking). A formal specification defines the required behaviour more precisely and with more attention to potentially problematic behaviour. The formal specification is an optional layer, as it is not required when developing software. When provided however, it can give guarantees about e.g., proper use of cryptographic protocols. Unfortunately, formal specifications can become outdated and might also only cover part of the application.

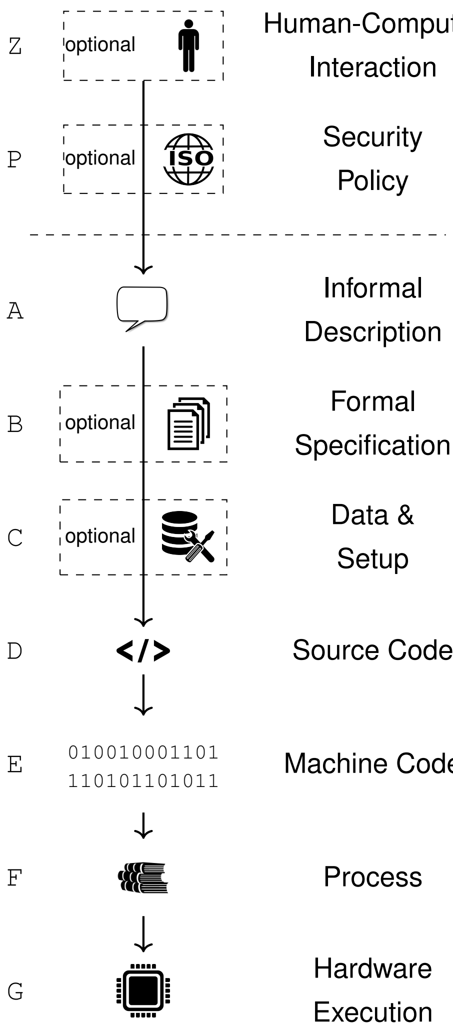


Figure 4.2: Diagram of the different abstraction layers in GEN. At the top it contains two meta-layers. The remainder shows the translation from a conceptual application to a concretely executed version. Discrepancies between different layers form a *bug*: a layer behaves differently than intended.

Formal specifications are always translated from informal descriptions. A vulnerability can therefore occur between layers A and B, since there are no guarantees that the formal specification maps perfectly onto the intended behaviour from the informal description. Discrepancies here are exceptionally problematic as their absence cannot be mathematically proven. We therefore cannot fully prove the absence of vulnerabilities within an application (i.e., absence of evidence is not evidence of absence).

C **Data & Setup.** Next is the data layer, also called the Software Setup layer. This layer is classically concerned with the configuration of the software. Consider here an SQL server that is exposed to the Internet under credentials `admin:admin`. This is an issue with the setup of the server, not the software itself. Note that this layer is optional since it is not a solid representation of the software, but instead specifies pieces on how the software will behave in practice — just like the formal specification layer. This also means that a vulnerability can skip this layer entirely when it does not involve the configuration, data or setup of the application.

Recent interest in Machine Learning (ML) highlights new attack vectors unaccounted for in related work. Through ML, *data* can determine the behaviour of an application rather than the code. This data-driven development requires us to reason differently about the data component and vulnerabilities in this new setting. As data can specify behaviour, the data layer considers behaviour as specified in an ML setting too. Due to the importance of data in this setting, two new vulnerability types arise when applying ML. Once a dataset has been poisoned, it no longer accurately represents the classes or clusters one intends to represent [110, 13]. This is a bug from a higher abstraction layer to the data layer. Note that we are not discussing the dataset poisoning itself, but instead the vulnerability that *arises* from an already poisoned dataset.[23]

The second attack considers an adversarial sample, where a data-point is carefully crafted to represent a different class from the actual class [148, 147, 152]. Categorising adversarial ML is a difficult problem and can generally be attributed to either the dataset or the processing / learning step in ML. GEN considers the dataset lacking in detail (i.e., definedness) for the correct classification of the adversarial

[23]Poisoning the dataset itself can be done in a multitude of ways and is technically a payload phase technique in the framework of Chapter 3.

sample. Hence, we label this a discrepancy between the data layer and the next layer – the source code.

D **Source Code.** The Source Code layer considers the application in the programming language(s) used to develop the application. This describes what a program is supposed to do on a detailed, less abstract layer than any of the above. A program can have multiple dissimilar representations (i.e., languages) of source code and can even have different source code representations (e.g., Cython translating Python code to the C language). Within this layer, we *only* consider the various internal artefacts of the application that have been written by developers (in contrast to e.g., auto-generated or transpiled code). Discrepancies between D and B (specifically incorrectness bugs) are commonly sought for using formal verification.

Up to (and including) this point, translations from one abstraction layer to another are generally performed manually. From here onwards, most aspects are automated: compilation, execution, etc. As a consequence, whenever a discrepancy occurs (due to incorrectness, as will be explained later on), this is often attributed to the external component responsible for that translation step. Intuitively, if the behaviour of a program changes after translation from source code to a compiled version, this means that the compiler incorrectly translates the source code.

E **Machine Code.** Naturally, the next abstraction layer is a compiled version of the code. We label this the Machine Code layer. Transformation into machine code is generally done automatically, through a compiler or interpreter. Common bugs in compilers arise from aggressive optimisation techniques. For example, when performing cryptographic operations, it is common to finish with zero-ing out the memory used for private data. Dead code elimination is known to remove this, since the newly written zeroes are not used before it goes out of scope [69]. The optimisation changes the behaviour of the application, making the resulting behaviour incorrect with respect to the source code layer.

The machine code layer considers all source code object files that were translated from source code (D) to the machine code layer (E). For *external* libraries, we require a separate layer since their source code is not part of the application.

F **Process.** As such, next is the Process layer, also known as the External Component or Library layer. This is the step where we consider our application as a running process on top of an operating system, in contrast to a static file (our executable). As process, we now consider segments instead of sections, we consider the implementation of libraries[24] and the process gains the ability to interact with the operating system. This layer considers the actual implementation of these components as behaviour, where previously we considered them from a specification point-of-view.

The process layer is an unusual layer, as we consider external components as a single layer abstraction. For each external component, the same layerisation from Figure 4.2 is applicable. Nevertheless, from the application point-of-view we merely care about its description (e.g., manual) and its implementation.

G **Hardware Execution.** When executing the machine code on a machine, we have the Hardware Execution layer. This is the "fully concrete" layer: it represents the execution of our application on an actual machine. Examples for vulnerabilities are the Rowhammer attack or the Plundervolt attack [164, 134]: these type of vulnerabilities get exposed only in hardware execution and are (incorrectness) vulnerabilities with respect to the above layers. Most side-channel attacks fall in this category too, as indirect output such as electricity usage becomes explicit within this layer.

P **Security Policy.** Above the informal layer, two layers exist that do not have an application representation, but rather contain meta-properties. The (Security) Policy or (Security) Standard layer describes a number of properties that should hold within the given application and its usage. The properties within the policy layer can vary across different other layers, and this layer can be seen as addition to the informal layer. For example, a standard requirement may be that all network connections must be cryptographically secured. If this ends up not being the case, it will result in a vulnerability. Alternatively, if left unspecified (e.g., because no such standard is included), a lower abstraction layer gains the power to decide if the connections will be encrypted or not. This would however *not* be a bug or vulnerability from the policy perspective, because the policy is not violated. Various different vulnerability (and attack) classification models distinguish between design, implementation and configuration. While our model roughly follows this

[24]Here we consider all libraries where the source code is not under our control — including static libraries.

```
Question> 5 / 2?
Answer> 5 / 2 = 2
```

(a) An *incorrectness* bug: the program is supposed to return 2.5.

```
Question> apple / 2?
Segmentation fault
```

(b) An *undefinedness* bug: it was not defined how the program should behave.

Figure 4.3: Example undefinedness and incorrectness bugs.

narrative, the addition of a security policy seems to have been overlooked in models we have encountered. Where an application is well constructed and deployed, it does not automatically mean that it follows the rules and regulations set on a bigger scale.

Z **Human-Computer Interaction.** Finally, we introduce the Human-Computer Interaction (HCI) layer. This layer considers the user-behaviour of the application, both directly and indirectly. We consider this layer to only interact with the informal description layer (A) and the policy layer (P). An example policy is to not re-use credentials for the application, and an example attack is a spear-phishing attack.

Although we added the HCI layer for exhaustiveness purposes, we consider this layer out-of-scope in the remainder of this dissertation. Treating this layer in the same way as other layers would specify what layer is "correct" (the human or the description/policy) and this is not as straightforward: it is a human issue and should be treated as such. Maybe even more concerning in treating this layer equally would be the search for such a vulnerability. The authors do not believe searching for vulnerabilities within users is the right approach ethically. We leave work on this issue domain experts in that area [2, 210].

4.2.2 The Incorrectness-Undefinedness Property

Once a discrepancy is observed between two abstraction layers, it raises the question of how this discrepancy came to be. Another tempting question is what layer is responsible or needs fixing. The answer to this lies within the *Incorrectness-Undefinedness property (IU property)*.

Incorrectness. Incorrectness occurs when two layers disagree about the behaviour of the application. Consider a program that expects a number as input and returns exactly half that input (i.e., $x \mapsto x/2$). Upon sending an odd number as input, say 5, an application using integers cannot return 2.5 and will likely return 2 (or 3).

This is considered inconsistent with the intended behaviour as given above: we say there is an *incorrectness* bug between the informal layer and the source code layer. Figure 4.3a illustrates this.

When faced with an incorrectness issue, we always consider the "higher layer" layer to be correct and the lower layer to be incorrect. After all, the higher layer describes the intended behaviour of an application. This eliminates any ambiguity (with respect to incorrectness bugs) as to which layer should be regarded as correct. Through transitivity, the elimination of an incorrectness bug will always end up being consistent with the top-level layer that represents the intended application.

Although an application may do what it is supposed to do, it may also have side-effects that introduce vulnerabilities (e.g., a backdoor). We consider these incorrectness bugs, as they do not *only* perform the requested operation. A more thorough discussion on side-effects is found in Section 4.8.

Undefinedness. The second principle that leads to a deviation is undefinedness. This is a more nuanced issue, as undefinedness means we have an *underspecification*: there is no expected behaviour. We are looking for the absence of something being defined, instead of something incorrect. Unfortunately, this is where most known vulnerabilities arise: through corner-cases that were not accounted for. The abstraction layer therefore leaves the behaviour undefined.

Given the calculator example above, our description leaves undefined how the application is supposed to behave when the input is not a number: See Figure 4.3b. A string as input has been left undefined and hence anything can happen in practice — a segmentation fault in this case. This issue is mathematically known as *Reductio ad absurdum*: False implies anything. We refer to this type of issue as an *undefinedness* bug. Note that we left the behaviour undefined when the input is not a number. If we had specified to return an error message on any other input (defining behaviour in the higher layer), either the error message shows, or it would be an incorrectness bug.

When undefinedness occurs in an abstraction layer, it could become defined in a lower layer. In fact, this is where undefined behaviour can become useful, as the layer leaving something undefined can simply assume it will not occur. This can simplify the translation from one abstraction layer to another. A simple example to demonstrate this is a buffer overflow in C. This is explicitly stated to be undefined behaviour in the C language specification to enable additional optimisation opportunities during compilation.

On the other hand, it is pivotal to the correctness of the application to never *execute* undefined behaviour. If we end up doing so, the *actual* behaviour could be determined by the compiler used or affected by non-deterministic factors (e.g., randomness or environment) — but defined nevertheless. The observed behaviour will be defined in a lower abstraction layer like the C compiler does in the example above.

We make note that a gap is filled during translation between specification and code, forcibly defining behaviour where it was previously left undefined. The actual behaviour can be anything at this point, as we have lost all assurances about current and any future behaviour of the application. In other words, giving the space to define behaviour within a lower abstraction layer is often problematic by design. In contrast to incorrectness, we consider the *higher* level layer to be incorrect and in need of fixing, as this layer executes undefined behaviour.

Finally, undefinedness should not be confused with hiding details through abstraction. Within our calculator example as discussed above, `apple / 2` was left undefined. When input is defined (e.g., `4 / 2`), the description of the program does not explain *how* to perform the division as these implementation details are left for lower abstraction layers. This however is *not* an undefinedness bug — in contrast to the apple example — as the operation itself is specified.

Defining bugs.
An important observation is that **incorrectness and undefinedness are naturally (1) mutually exclusive and (2) exhaustive**. If behaviour has been properly defined, this can never trigger an undefinedness bug. Similarly, if behaviour has not been defined at some point, it can technically not be incorrect either. Additionally, if behaviour of an application is properly defined and the application follows the exact specified behaviour, there's no room for bugs.

This property provides us with enough information to formulate a technical definition of a bug. Concretely, we define a bug as follows:

Definition 1. *A software bug is a deviation between intended and actual behaviour that occurs when two abstraction layers have a discrepancy. This is due to either (1) incorrectness or (2) undefinedness.*

This definition contextualises the common definition of a bug as a mistake in software, making it a quantitative and descriptive property. On top, it provides additional information about the bug.

4.2.3 From Bugs to Vulnerabilities

So far, we described how GEN defines a *bug*, with an abstraction layer discrepancy and an underlying cause (i.e., incorrectness or undefinedness). We next discuss when GEN considers a bug to be a *vulnerability* (Figure 4.1, step 3). In GEN, we consider the core difference between a bug and a vulnerability resides within the *assets* of the computer system, namely its capabilities and data. As such, we define a vulnerability as a bug that could expose *previously inaccessible capabilities or data* on the target system:

Definition 2. *A software vulnerability is a bug (as per* DEFINITION *1) that can potentially expose data or functionality that was not intended according to the informal description or policy.*

Although the definition focuses on misbehaving software, this does not imply that bugs or vulnerabilities are limited to software itself. Hardware, micro-code or social issues become bugs due to the resulting software misbehaving.

Note that according to this definition all vulnerabilities *are* bugs.

This definition of a vulnerability refers to some unknown factor: a potential rather than a certainty. This is because assessing the (un)exploitability of a bug can be a tremendous task [70], so it is often not worth the effort to determine this with certainty. Even heuristically, it is hard to assess the exploitability of bugs as the most harmless bugs can potentially be leveraged in a complex attack. For example, a null pointer deference might trigger the restart of a process which in itself seems fairly harmless. Yet, it can make the heap layout (more) deterministic and limit noise [92]; and reset the ASLR offset or the stack canaries when it may have unwritable characters (e.g., null bytes). This can be argued to expose multiple capabilities: it might not be so harmless.

Note that our definition does not mention the property of exploitability. The exploitability property is not objective, as the leaking of some internal data may be useful to an attacker but this is far from guaranteed. An example is CVE-2014-0160, better known as Heartbleed [51]. Heartbleed is a buffer overread vulnerability in OpenSSL that can leak adjacent memory, including a private key, as will be discussed in more detail in Section 4.3.2. Yet, without any sensitive data within proximity to the overread bug, the newly obtained data by the attacker could have been completely useless. In other words, it is non-trivial and highly context-sensitive to generically determine the sensitiveness of a bug in more detail. This would become a *severity* assessment, which is out of scope for GEN.[25] We consider low severity bugs to still be able to impact the security of a system. Thus, we suggest defaulting this property to true unless *certain* the bug cannot provide any new data or functionality. Concluding, the third step of bug classification in GEN is not answering whether the bug could expose new capabilities or data, but instead whether we have some level of confidence that the bug does *not* expose new data or capabilities.

The GEN definition of a vulnerability is embedded within the GEN design and thus very different from the definitions as discussed in Section 4.1. Where existing vulnerabilities start by questioning the effects (e.g., exploitability or security policy violations), our definition is grounded in its origin: in why the vulnerability came to be. What mistake was made for this vulnerability to be exposed lies at the center of GEN — how they affect security principles is of secondary concern.

Because the definition is fundamentally different, it is difficult to summarise the GEN definition in the same way as we did in Table 4.1. Nonetheless, we would argue to fall in the categories of potential exploitability (directly following the discussion above) and affecting assets (following directly from Definition 2). The vulnerability *could* arise from the security policy (layer P) but does not need to do so: plenty of vulnerabilities exist that may not be taken into account in the policy. We also do not mention a threat source or threat actor, as a lack of an explicit threat does not remediate the situation. Finally, it can be argued that we do consider confidentiality, integrity and availability through the definition of exposing data (confidentiality) or new capabilities (integrity, availability). Either way, no existing definition is similar, as Table 4.1 reflects.

[25]For a severity assessment, other frameworks are available, e.g., CVSS [162].

4.3 Applying GEN to a Vulnerability

In Section 4.2 we introduced the conceptual model underpinning GEN. This section describes how GEN is used to classify vulnerabilities in practice. We begin with a discussion of how GEN assigns a *class label* to vulnerabilities. Then we discuss challenges that arise when applying GEN and how those challenges can be resolved. A detailed classification sheet with guide can be found in Appendix A.

4.3.1 GEN Class Labels

To classify a bug with GEN, we follow the three steps in Figure 4.1. In step one, we question between what abstraction layers the bug manifests. Afterwards, we see if behaviour is specified on the higher abstraction, i.e., whether behaviour is due to incorrectness or undefinedness. Finally, we assess whether it has an impact on the data or functionality of the system, to determine whether it is a vulnerability or not.

Using the properties discussed, we can label each bug with a given type. For this, we use the following format: `[higher layer][lower layer]-[IU property]`. Within the layers, we can either give the full name (e.g., "Source") or the unique character (e.g., `D`). The IU property is either I for incorrectness or U for undefinedness.

As an example, a classic buffer overflow is undefined behaviour according to (most) source programming languages where this is possible to occur. Hence, it is a discrepancy between source code (`D`) and the machine code layer (`E`), where the behaviour becomes specified (see Table 4.2). Note that it may not be deterministic yet, but defined nonetheless. According to our notation rules, the correct GEN code would be "Source-Machine-U" or "`DE`-U" in short. As such, the heap overflow from our example in Section 3.1 is a "`DE`-U" vulnerability.

4.3.2 Classification Challenges

When classifying a bug with GEN, we have identified three situations that introduce conceptual challenges for the GEN model where additional care is required in practice: (1) fixing bugs can expose or remove other bugs; (2) an undefinedness bug that was once defined creates a three-layer pattern; and (3) the informal layer could

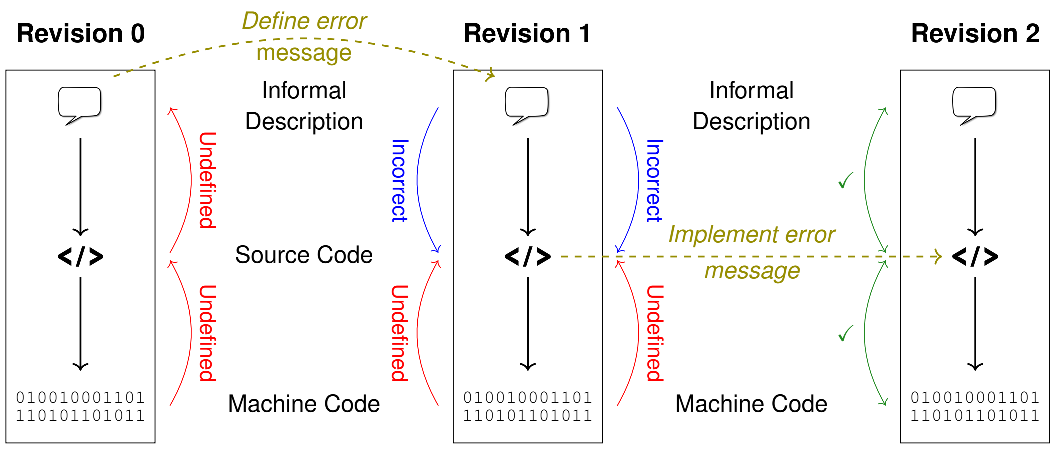


Figure 4.4: This diagram depicts how `GEN` tackles the undefinedness bug from Figure 4.3b, where non-integer input is not defined. First, we fix the undefinedness issue to show an error message upon non-integer input. Secondly, this needs to be implemented in Revision 1, and the problem is resolved completely in Revision 2.

contain contradictory statements. We next discuss when each occurs and how they can be overcome.

Fixing bugs can expose or remove other bugs related to the same problem. As an example, lets reconsider our undefinedness example from Figure 4.3b, where $2 + \texttt{apple}$ was undefined. The informal description does not specify how to handle this, so there is an undefinedness discrepancy from the informal description to the machine code layer — where it becomes defined. Upon defining behaviour (e.g., defining that any other input should show an error), the source code does not yet reflect this as it is still undefined here. This interaction is shown in Figure 4.4, while going from Revision 0 to Revision 1. Unfortunately, this fix presents us with an incorrectness discrepancy, as discussed next.

As long as the related bugs are of the same IU type (see the next section on incorrectness through undefinedness), fixing bugs should be done from the top-layer down. Incorrectness bugs could introduce or eliminate related incorrectness bugs in the layers below, while a lower layer fix could end up incorrect after fixing the higher layer incorrectness. With undefinedness bugs we do not even have a choice: if multiple layers leave the behaviour undefined, we need to find the first layer that contains defined behaviour. If no layer provides defined behaviour, the informal description needs to specify the behaviour and the bugs can be fixed from there onwards.

Incorrectness through undefinedness occurs when a lower abstraction layer (hypothetically called "V") contains undefined behaviour that was initially defined behaviour in a higher abstraction layer "U". It becomes defined again in a lower layer "W", such that we end up with a defined-undefined-defined chain of relationships. In the first half — defined-undefined — this is an incorrectness bug. In the second half — undefined-defined — it is an undefinedness bug instead. Due to the high layer defining behaviour, this creates a two-fold classification, generally with a single bug in layer V.

This is shown in Revision 1 of Figure 4.4. Because the informal description is updated, there is now an incorrectness discrepancy between the informal layer and the source code layer. Similarly, an undefinedness discrepancy exists between the source code and the machine code layers with respect to the same overall problem. Implementing the newly defined behaviour fixes *both* the incorrectness discrepancy and the undefinedness discrepancy.

For a more extensive example, lets reconsider Heartbleed [51]. Heartbleed is a bug that appeared in the heartbeat mechanism of the OpenSSL library. The heartbeat mechanism can be used by clients to check for a response, by sending a small payload and providing the length of the payload in the `payload_length` field. The server would then respond with the given payload and length, confirming the connection between client and server. The Heartbleed bug occurred when the `payload_length` passed to the server was larger than the data being passed on. In the vulnerable versions of OpenSSL, the server would respond with `payload_length` bytes, even if the payload given was shorter. The additional bytes sent would then be taken out of adjacent memory, where private keys could reside. This is an incorrectness bug, as the specification says to either (1) respond with an exact copy of the request, or (2) silently discard if the `payload_length` variable is too large [101]. On the other hand, it triggers an undefinedness bug in the Source layer through a buffer overread. After all, reading outside a buffer in the C language is undefined behaviour.

Undefinedness bugs always trigger an incorrectness bug, unless it was undefined in previous abstraction layers (in which case that needs to be addressed separately).

In an incorrectness-through-undefinedness bug such as Heartbleed, we conclude that it has two classes, since it refers to the same underlying bug. Alternatively, an undefinedness bug leading to another undefinedness bug (in a higher abstraction layer) is considered separate as multiple layers have a bug that needs to be resolved. Once resolved, this will lead to an incorrectness bug paired with the original undefinedness bug. In other words, an undefinedness bug is either (1) at the top abstraction layer, (2) automatically introduces an incorrectness bug on the layer above, or (3) contains additional, separate bugs that need to be resolved that would lead to situation 2. Because of this property of undefinedness bugs (in contrast to the simpler incorrectness bugs), we can label them by their undefinedness characteristic — the incorrectness bug will always be there until the undefinedness bug is resolved.

The problem when failing to address the bidirectional nature of undefinedness bugs is clearly shown in the CWE classification [129]. They do not address this and end up with ambiguous classification where e.g., a buffer overflow (CWE-120) can also be classified as Improper input validation (CWE-20). Note that these two CWE classes have no relation according to the CWE. More so, a problematic buffer overflow could consequently lead to a write-what-where condition (CWE-123), labelling the vulnerability with 3 CWE classes. We will discuss the interaction between GEN and CWE in detail in Section 4.4.

Intra-layer Contradictions can happen in a very exceptional case: it could occur that the informal description by design could contradict itself. For example, a description of the form "Button 2 must perform operation B; Button 2 must perform operation C" (where operation B is different from operation C). In this case, we have to consider the informal layer to be incorrect with respect to itself. This can only occur in the informal layer and in exceptional cases the policy layer. Any other layer must in the end conform to the informal layer.

4.4 The Common Weakness Enumeration

The Common Weakness Enumeration (CWE) [129] is an operational vulnerability classification effort made by MITRE and the most common classification method currently in use to the best of our knowledge. The CWE is fundamentally different from GEN in a number of ways. This section highlights the differences between CWE and GEN, finding out how a combined usage of both could enhance classification.

CWE's bottom-up approach means an observed vulnerability is generalised into a category. [26]. The drawbacks of this approach are that it provides no guarantees in terms of exhaustiveness or disambiguation as the model requires continual updating to cover new forms of vulnerabilities, as shown in Table 4.2. CWE has a total of 418 classes at the time of writing, with new classes being added over time.

In contrast, GEN follows a *top-down* approach. By design, GEN has limited ambiguity in comparison to CWE and provides a reasoning structure that can describe the entire spectrum of bugs – including new bugs that could arise in future. On the flipside, we do not argue that a GEN label like *Source-Machine-U (DE-U)* is as descriptive and intuitive when compared to a CWE label like *classic buffer overflow*. GEN compensates for the limitations of CWE. To understand how CWE and GEN interact in practice, we mapped *all* CWE classes onto one or more GEN classes. Besides showing interactions between the two, the mapping can also help classifying with one, given the class of the other classification. A few example vulnerabilities with both classes are shown in Table 4.2.

Within the mapping, 41% of CWE classes have a single GEN class. In 39% of cases more than one GEN class is attributed to a CWE class. We refer to such CWEs as *multi-class* CWEs. Conversely, the remaining 20% of CWE classes are classified as *pseudo-vulnerabilities*, meaning the CWE class does not represent a bug according to GEN. We next discuss both multi-class CWEs and pseudo-vulnerabilities in more detail.

Multi-class CWEs. Many of the CWEs encountered had multiple associated GEN classes. This phenomenon can have two different yet closely related reasons. In the first case, the CWE description itself reflected this directly, stating two or more situations belonging to the CWE class. As an example, CWE-241 describes the issue when "the software does not handle or incorrectly handles [input]" of an unexpected data type. In the case of an unexpected data type, it is likely to be undefined within the informal description (A). Hence, when the software does not handle this type, it will be classified to Informal-Source-U. On the other hand,

[26]For a more detailed description of the CWE, see Chapter 2

Name	CWE class	GEN	Code	Explanation
Buffer Overflow	CWE-120	Source-Machine-U	DF-U	Undefined behaviour in common languages like C/C++ becoming defined behaviour after compilation.
SQL Injection	CWE-89	Informal-Source-I	AD-I	One would expect to perform the query as it would with proper sanitisation: the source code fails to do so semantically.
Rowhammer Attack	CWE-1256	Machine-Hardware-I	FG-I	Flipping adjacent bits in memory should have no impact on the state of the application. Here the hardware fails to adhere to this property.
Password Sharing	CWE-522	HCI-Informal-I	ZA-I	If specified that password sharing is prohibited, given fair reason and usability as such, the human does not adhere to the rules as they should.
Dataset Poisoning	N/A	Informal-Data-I	AC-I	The dataset no longer accurately represents what it should, leading to incorrect behaviour compared to the intended behaviour.
"no-s"ftp	CWE-319, CWE-284	Policy-Data-I	PC-I	If specified that (TLS-)encryption must be used, usage of plain FTP is incorrect, even if the informal description is not explicit.

Table 4.2: Typical vulnerabilities with their GEN class and an explanation as to why they belong in this class. Each class consists out of two abstraction layers and either an I for incorrectness or a U for undefinedness.

when the software *incorrectly* handles the input, it has to be defined how to handle this input and has to be an Informal-Source-I type bug. Concluding, CWE-241 has two classes: AD-I and AD-U. So a single CWE description can in effect conflate two or more different GEN classes.

The second case occurs when a detail is unknown, meaning the CWE class could fit into multiple GEN classes. For example, an integer overflow (CWE-190) in the C language is undefined behaviour (hence DE-U) when the integer is signed, yet defined behaviour in the case of an unsigned integer (hence AD-I or no bug at all). If instead the source language is Java, a signed integer overflow would be defined as rollover behaviour and thus not an undefinedness bug either. Do note that these two are *fundamentally different*: integer overflow behaviour can be used by the developer if defined, but should **never** be used if its behaviour is undefined. One compiler may behave as the developer expects, but a different compiler or compiler version may not [170].

Missing detail that leads to multiple classes can be very different in nature too. CWE-321 discusses the use of a hard-coded cryptographic key. When this is set as configuration of a program, it is a Policy-Configuration-I bug. Alternatively, it could have been hardcoded inside the source code of the application, where it would be a Policy-Source-I type bug. This lack of detail in CWEs can lead to 3 or 4 classes in exceptional cases.

Pseudo-vulnerabilities. In total, 83 CWE classes were classified as pseudo-vulnerabilities: classes that did not represent real vulnerabilities. This is attributed to a number of different reasons. First, CWE has a number of classes that represent *consequences* (such as a write-what-where, CWE-123, that states an attacker has some level of control over inter-application virtual memory) rather than the vulnerability itself. This is likely the result of a vulnerability, but is not a representation of the vulnerability itself. Consequences cannot be classified with `GEN` because it has no information on either property `GEN` requires for classification.

Secondly, CWE contains classes that are *indicators* for vulnerabilities, such as Suspicious Comment (CWE-546). Similar to the consequences, these might be useful in an operational setting but should not be confused with an *actual* bug or vulnerability. These cannot be classified either because there is no incorrectness or undefinedness occurring through the indicators.

Last, *"bad" code* that leads to sub-optimal performance or redundancy (affecting maintainability) is also classified as a vulnerability according to the CWE classification. Again, this is not a vulnerability according to `GEN` unless the security policy layer explicitly states this is not allowed. Only the CWE classes here mentioning a security policy were classified with the appropriate policy class(es).

CWE & `GEN`. `GEN` is a top-down approach and hence fundamentally different from the bottom-up CWE. It captures properties of vulnerabilities that CWE does not capture in its current form. The mapping as discussed substantiates this claim, as fewer than half of the CWE classes conform to a unique `GEN` class. On the other hand, concreteness and descriptiveness of CWE classes provides a quick intuitive

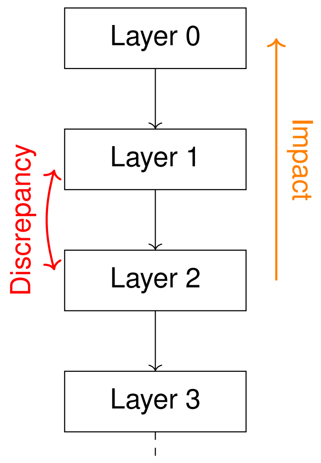


Figure 4.5: This figure visualises that the impact of a given vulnerability can only flow upwards from the lowest affected layer in the abstraction hierarchy. The layers could represent any `GEN` layer subset (e.g., layer 0 being the informal layer, layer 1 being the setup layer, etc.).

understanding that `GEN` cannot grasp. Thus, we believe `GEN` can complement CWE in the classification of given vulnerabilities, similar to how CVSS complements CVE. `GEN` can also help provide the right CWE class(es), because a `GEN` class only has a subset of related CWE classes.

4.5 Using GEN

Having introduced the conceptual model underpinning `GEN` (Section 4.2) and how it can be applied to a vulnerability (Section 4.3), in this section we discuss the use of `GEN` beyond its theoretical framework. First, on an individual vulnerability level we discuss how `GEN` bounds the extent to which a vulnerability can impact system behaviour. Second, we discuss how different `GEN` classes relate to different testing techniques. Specifically, we describe how to find vulnerabilities of a given `GEN` class and the construction of new, specialised testing techniques based on the `GEN` methodology. Finally, we discuss a direct practical use-case of `GEN`.

4.5.1 Consequence Boundaries

GEN's abstraction layers indicate where vulnerabilities occur. Although the severity of a vulnerability is out of scope for this work, these layers can also provide insight in what might be affected as a consequence of the vulnerability being abused. That does however not mean that a vulnerability cannot surpass its abstraction layer boundaries.

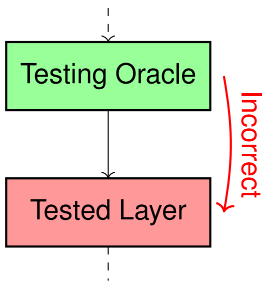
In particular, vulnerability consequences can *move up* abstraction layers, but *cannot move downwards*. This is because complexity and detail is added when translating to lower abstraction layers. Vulnerabilities on a higher abstraction level cannot affect the lower level details, whereas a lower level vulnerability can still impact the overall operation. This is visualised in Figure 4.5 Concluding, GEN can only aid in filtering lower-level consequences, but not the converse.

As an example, consider the process of compilation: moving from the source code layer to the machine code layer. One such vulnerability is a buffer overflow, that falls in the `Source-Machine-U` category. A buffer overflow can alter the internal memory state of an application, which can in turn affect the operation in a semantically significant manner (e.g., crashing the application). The converse is not true: a vulnerability on a higher level (say an SQL injection) will not be able to impact e.g., the internal memory state. This is because the vulnerability resides in an abstraction layer too high and hence has no concept of internal (virtual) memory.

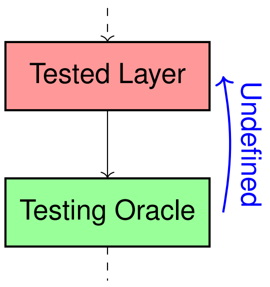
4.5.2 Abstraction layers as Testing Oracle

GEN's design is based on the way we try to find different bugs and vulnerabilities. GEN can explain why fuzzing an application with a commodity fuzzer is unlikely to find a Rowhammer vulnerability [196, 164]: the fuzzer has no knowledge about hardware-specific properties. Similarly, searching for a Rowhammer bug or using a fuzzer will not expose a server with `admin:admin` credentials. In this section we discuss how GEN is related to bug-finding techniques and how a new bug-finding technique can be designed using GEN concepts.

According to GEN, when two abstraction layers misalign it must be due to either incorrectness or undefinedness. In the case of incorrectness, GEN considers the *higher* layer to be correct. In other words, if we can test the lower layer based on the behaviour specified by the higher layer, we should be able to find the bug. The



(a) With incorrectness, the upper layer can be used as testing oracle to test the lower layer and find such bugs.



(b) We can find undefinedness discrepancies by recognising undefinedness artefacts in the lower layer, potentially after altering the lower layer.

Figure 4.6: Incorrectness and undefinedness work in opposite ways when searching for such bugs. With either type of discrepancy, one of the involved layers can be used as testing oracle to test the other involved layer. The green layer depicts the layer that can be used as testing oracle, while the red layer depicts the layer that we are testing.

higher layer in this instance can be called a testing oracle [11]. We can use this abstraction layer as ground truth to test correctness of the lower layer, in the traditional sense of a testing oracle. For example, a formal specification is a testing oracle for the source code layer w.r.t. incorrectness issues. Concretely, any incorrectness bug of a given GEN class can be found when using the higher layer as ground truth as shown in Figure 4.6a. For example by using any existing technique that does so.

Construction of Undefinedness Oracles. In contrast to incorrectness, when two abstraction layers misalign due to undefinedness the higher layer is in the wrong. Hence, we must use the *lower* layer as testing oracle, because we must test the higher layer to determine if and where it is underspecified. This is shown in Figure 4.6b. However, we need to *build* this oracle from the lower layer, as it provides a change from undefined behaviour into defined behaviour: a discrepancy in behaviour that we can pick up on.

We can build a testing oracle for undefinedness bugs in a number of ways. First, we can *define the undefinedness* with a marker. A concrete example is UBSan, an undefined behaviour sanitiser for C-like languages [183]. Here we force any undefined behaviour in the source code to throw an exception. The exception is the marker we can use to uniquely recognise whether undefinedness occurred. Through this we can test the layer below (the machine layer) and detect any undefined behaviour.

Another similar solution is to prepare for such bugs by design, and define problematic situations as *should not occur*. A very common example is the addition of `assert` statements by developers. Similarly, an application design (informal description) could specify that a specific capability should never be visible if the user does not have sufficient privileges. This solution does not cover all vulnerabilities of a given type but can be used to build up an approximate undefinedness testing oracle.

Thirdly, we can find undefined behaviour through *expected patterns or artefacts*. Similar to the first solution, we can consider source code undefinedness bugs. A common pattern here is for undefinedness bugs to cause a crash. As a reminder, this is where buffer overflows reside, among others. In any layer below the source code, where translation is generally automatic, undefined behaviour can damage internal structures, leading to crashes more often than not. In any higher abstraction layer — where the translation is not automatic — alternative patterns should be found. The expected patterns will never guarantee complete coverage of all vulnerabilities within that class however.

Lastly, we can abuse the bidirectional nature of undefinedness, since the layer above (when it exists) either contains another undefinedness bug with respect to the layer above, or contains an incorrectness bug (since we go from defined to undefined). In the latter situation, we can test for the incorrectness bug instead, if the actual behaviour changes the expected behaviour due to the undefinedness bug. This can however not be used when undefinedness is purposefully used (such as assuming the memory layout during run-time) if the actual behaviour ends up matching the intended behaviour due to other factors.

All of the above are valid techniques for creating a bug-finding technique for a given GEN class. Using this in conjunction with the incorrectness issues where we can take the higher layer as oracle, we can determine what bug-finding technique is searching for what GEN class bug.

In Practice. In some cases, the abstraction layer is a testing oracle in the traditional sense. When performing *formal verification* (e.g., through model checking [41]), the formal layer *is* a testing oracle for the source code layer. In practice, most other bug-finding techniques approximate a testing oracle with respect to one or two different classes, but have the potential for finding bugs in other classes. Below we will discuss a few examples of this.

Techniques such as *fuzzing* [31] generally search for application crashes and hangs. Crashes and hangs are almost certainly problematic, and we refer to them as an implicit oracle [11]. These crashes (and occasional hangs) are typical artefacts of source code layer undefinedness. Note that fuzzing does not look into hardware-specific attacks (unless tailored towards these), and often crashes and hangs are not explicit in source code. Even more so, common practice is to not instrument (and hence not enhance code-coverage in) dynamic libraries in order to prioritise the application. In other words, we hypothesise that most bugs found within commodity fuzzers (that trigger upon crashes/hangs) fall within the Source-Machine layers, with an additional few within the Machine-Process layers. The testing oracle becomes exact when a fuzzer is combined with test-time sanitisers such as UBSan that can crash the application when undefined behaviour occurs in the source code[27].

Within the Source-Machine layers, there are also incorrectness bugs. As discussed before, an incorrectness bug in this layer refers to a mistake by the compiler. D'Silva et al. discuss this thoroughly, with the classic example of optimising out `memset(buffer, 0, sizeof(buffer))` at the end of a function [69]. The incorrectness bugs commonly seen in compilers nowadays do not induce hangs or crashes, so fuzzing will not find such bugs. Instead, it finds undefinedness bugs. Considering the `C` language for example, many vulnerabilities are known to arise from explicit undefined behaviour, e.g., buffer overflows, signed integer overflows and format string attacks. We test our fuzzing hypothesis in the evaluation (Section 4.6.2).

Once we get to the Hardware layer, attacks become sufficiently diverse that a single bug-finding technique is insufficient with currently available techniques. Instead, most types of vulnerabilities require a specialised technique to detect a particular vulnerability. In the search for vulnerabilities in this layer, we will discuss two different examples. If any Instruction Set Architecture (ISA) contains undefined or even incorrectly behaving instructions, specialised opcode fuzzing can bring this to light [67]. The second example — the Spectre/Meltdown family — is a result of an incorrect branch prediction. The incorrect (and resolved) branch

[27]As has been done before: https://groups.google.com/forum/#!topic/afl-users/GyeSBJt4M38.
[28]http://sqlmap.org

Testing tool/technique	Tailored towards	Exact Oracle	Generic
Fuzzing	DE-U	×	✓
Model checking	BD-I	✓	✓
Spectector [87]	EG-U	✓	×
N-version Programming [125]	AD-I	✓	✓
Penetration testing	PC-I/PD-I	×	✓
Manual testing (job title)	AD-I	×	✓
Unit/Integration tests	AD-I	×	✓
User observation	ZA-I	×	×
Buffer overflow detection [121, 123]	DE-U	✓	×
Rowhammer detection [105, 209]	EG-I	×	×/ ✓
SQLMap[28]	AD-I	×	×

Table 4.3: Relation between bug-finding techniques/tools and associated GEN classes. *Exact Oracle* refers to the use of the abstraction layer as oracle; *generic* means it can find more than one particular type of bug. In Rowhammer Detection, the second technique cites is generic whereas the first is not.

prediction generates data without clearing as a side-effect. The data that is not cleared presents the opportunity to read out this data — a side-effect that was not part of the specified operation. A listing of bug-finding techniques and tools with their respective GEN class is summarised in Table 4.3.

Testing oracles by *designing* abstraction layers. Conversely, we can design new specialised bug-finding techniques based on GEN principles. This is done by slightly adjusting abstraction layers to model a specific vulnerability type. This new abstraction layer can be compared against the original abstraction layer. Whenever the two abstraction layers exhibit different behaviour, it can only be a result of the additional modelling.

One example is Spectector [87] by Guarnieri et al. They use symbolic execution to find Spectre-related bugs. They do so by modelling the Spectre vulnerability in their symbolic execution engine. The created model represents a slight change in the abstraction layer used by the engine, only adding Spectre specifics. Afterwards, they compare an execution in this model against an execution without the Spectre model. Any deviations can only occur because of the added Spectre specifics. In GEN terminology, they used the symbolic execution engine as abstraction layer and created a new abstraction layer by only modelling Spectre behaviour. Afterwards, they tested the new abstraction layer, using the original abstraction layer as testing oracle.

Here we view each translation between abstraction layers within GEN as many small step translations. Needless to say, these smaller translations can become very complex, as the order of smaller translations can make a big difference. Furthermore, since we do not look into a full translation into a naturally stable abstraction layer, the small step translations need to be chosen with care. Still, this methodology can be generalised using fine-grained changes with the GEN methodology, isolating more fine-grained vulnerability classes.

4.5.3 Security Use-case

Here we will discuss a concrete hypothetical use-case scenario to show the concrete use of GEN as a taxonomy. Specifically, we will first discuss the classification of a single vulnerability. We then explain how to use of GEN in a larger, statistical setting. We do so by outlining a scenario involving a hypothetical company and their response to the GEN classification results.

A New Vulnerability. Consider a large company (referred to as The Company) that internally develops software for the services they provide to their clients. The Company has a Security Operations Center (SOC) and an internal IT infrastructure. The SOC is alerted on a Friday morning as a zero-day exploit is discovered inside the database software they use, e.g., CVE-2014-2669. It is known that the exploit was discovered as it was used in-the-wild and it is unknown whether The Company has been targeted too.

Within its first report, it is explained that the exploit contains multiple previously unknown vulnerabilities in the software and people are working on a patch. Both vulnerabilities are niche and complex mistakes in a part of the software that is hardly used in practice. This vulnerability has a CWE-189 tag, referring to “Numeric Errors”. The report makes a quick mention that both bugs are of the DE-U type: an undefinedness discrepancy between the source and the machine code.

The SOC quickly concludes that the machine can only be compromised at the permission level of the SQL server based on the information given. However, the SQL server application itself *can* be compromised, including its internal data structures. Before investigating whether The Company was targeted or how the vulnerability affects the application, the SOC team turn on a version of the SQL server hardened with full memory safety (e.g., using Hardbound/Softbound [64, 136] and CETS [137]) and turn off the original process. The server is temporarily slower than usual until a patch is published and the original server can be updated and reinstated. Now the SQL server experiences minimal downtime, the threat has been quickly thwarted and the SOC team can focus on investigating if there is any reason to believe they had been attacked already.

The Bigger Picture. The Company is also monitoring every vulnerability that is found in their software. When a vulnerability manages to bypass the testing phase of internal application development, it is recorded on a separate vulnerability list for subsequent root-cause analysis. Over the past 2 months, the application security team of The Company has seen a small spike inside the vulnerability list in the `DE`-U class. In the meantime, fewer `DE`-U seemed to have been found during testing over those two months.

The team knows `DE`-U type bugs are most often found inside their symbolic execution engine, a custom-built project altered to find bugs and optimise code coverage inside their software. Upon further inspection, the team realises that 2 months ago the company updated their compiler. The total speed-up of their internal software was substantial but the machine code changed significantly, rendering their symbolic execution engine mostly useless in its current setup. The team started optimising the symbolic execution engine for this new environment and almost instantly found another 4 `DE`-U type vulnerabilities.

4.6 Evaluation

We tested `GEN` in two different ways. First, we evaluate how well `GEN` does as a taxonomy in a theoretical setting. For this we use the taxonomy evaluation criteria listed by Derbyshire et al. [63]. Although originally defined for *attack* taxonomies rather than for vulnerabilities (where an attack can be a complex structure comprising multiple vulnerabilities) we believe the criteria given are universal enough for use in evaluating `GEN`.

Secondly, we evaluate if the theoretical concept of `GEN` classes as testing oracles is applicable in practice. To test this, we took one bug-finding technique and clas-

No.	Criterion	Sub-criterion 1	Sub-criterion 2
1	Accepted	Appropriate hierarchical format	Sensible and tangible variables
2	Complete & Exhaustive	Top-most categories sufficiently high level	Clear processes for adding categories
3	*Comprehensible*	Categorisation well-defined	*Descriptive and industry-congruent*
4	Mutually Exclusive	Sensible and tangible variables	One class excludes other classes
5	*Repeatable*	*Different people provide the same class/label*	No terms require interpretation
6	*Terms well-defined*	No terms require interpretation	*Descriptive and industry-congruent*
7	Unambiguous	All [bugs] can be categorised	Appropriate hierarchical format
8	Useful	No catch-all category	Only categorises [bugs/vulnerabilities]
9	Versatile	Clear processes for adding categories	Clear processes for adding *to* categories
10	Human representative	Contains human-focused category	Can categorise socio-technical issues

Table 4.4: The 10 criteria for taxonomies according to Derbyshire et al.[63], adapted to a vulnerability taxonomy. The italic criteria represent the criteria that could not be sufficiently confirmed.

sified the vulnerabilities found. Here we used AFL [207], a fuzzer famous for its practicality and ability to find real-world bugs. According to our theory, the vast majority of the classified vulnerabilities should have the DE-U class, as this would be in line with Section 4.5.2.

4.6.1 Taxonomy Evaluation Criteria

For a fundamental taxonomy, we aimed to adhere to fundamental properties. For example, a bug should get *exactly* one label and *if* you can label it then it must be a bug. To evaluate the fundamental basis of the GEN taxonomy, we assessed it using the criteria specified by Derbyshire et al. [63]. They identify 10 criteria with two sub-criteria each to assess a taxonomy.

The taxonomy criteria are aimed towards attack taxonomies rather than vulnerability taxonomies. Although similar, the differences change the meaning of some criteria and also their importance. Although this requires a degree of judgement, to enable reproducibility we present for each criterion the detailed analysis underpinning our evaluation. The criteria with subcriteria are summarised[29]in Table 4.4.

Analysis. The first criterion is called *Accepted (1)* and requires both an appropriate

[29]The changes to the criteria are marked with straight brackets ([]) and only changes the focus from attacks to bugs/vulnerabilities. Furthermore, all criteria are left as is.

hierarchical format, and tangible and sensible variables used for categorisation. GEN is hierarchical along three dimensions: abstraction discrepancy; the IU property; and potential exposure of data or functionality. Each dimension can be used independently for comparison, similar to Engle et al. [75]. We believe this form of hierarchy is versatile enough to reason about similarities amongst bugs and vulnerabilities given their respective individual properties. Furthermore, each is based on a single tangible variable apart from the last. Abstraction discrepancy is tangible as it comprises the complete stack of software development and usage. The IU property is tangible too because any unclear situation is a result of ambiguity, meaning it is not well-defined (i.e., undefinedness). However the potential exposure attribute of our vulnerability definition (Section 4.2.3) can be difficult at times, but it defaults to true upon doubt.

Second is the criterion of *Exhaustiveness (2)*. Exhaustiveness requires the top levels of a taxonomy's hierarchy to be sufficiently high-level to capture all possibilities. The abstraction layers as presented in this work are exhaustive, ranging from the most abstract version of the application to a fully concrete version. Furthermore, the IU property is exhaustive: if neither occur, the translation is perfect. The vulnerability property is a yes/no question where any other answer would be equivalent to yes and is hence exhaustive. The second sub-criterion for exhaustiveness is clarity when categories need to be added. Within our categories, this should not occur because of our top-down approach to tackle the exhaustiveness issue. However, abstraction layers can be adjusted when appropriate to capture clearer subcategories. This will be discussed in Section 4.8.

The next criterion is *Comprehensibility (3)*. Whether GEN meets this criterion is not clear-cut, since the GEN classification is novel and requires a slightly different way of thinking about a given vulnerability. Specifically, Derbyshire et al. [63] specifies three sub-criteria that need to hold: industry congruence, descriptiveness and well-definedness. At this stage GEN has not yet been shown to be industry congruent, since it contains novel concepts unexplored until now. On the other hand, we argue it is descriptive (see Section 4.2) and its properties are naturally well-defined since the abstraction layers conform stable states in an applications' representation and the IU-property is both exhaustive and mutually exclusive.

Mutual exclusiveness (4) is very important in a taxonomy. If a vulnerability could reside in more than one category, it could cause confusion that would completely defeat the purpose of the taxonomy. Whilst we achieve mutual exclusiveness, it did come at a cost: we introduce separate vulnerabilities if bugs span multiple abstraction layers. As an example lets reconsider the calculator undefinedness bug from Section 4.2.2. As depicted in Figure 4.3b, `2 + apple` is left undefined in the informal description layer. The behaviour is most likely still left undefined in the source code layer, so the bug initially can be attributed to the DE-U class. Upon resolving this, the informal description still leaves this behaviour undefined, but the source code does not anymore: this is a clear undefinedness bug again. Now we consider the other AD-U bug *a separate* bug from the initial bug. Although counter-intuitive, it requires resolving bugs on multiple layers so we consider them separate bugs. If this approach is adhered to, mutual exclusiveness is achieved.

The next criterion states that the taxonomy needs to be *Repeatable (5)*. From a theoretical perspective, the abstraction layers are in line with the Time of Introduction property, an established property for classifying vulnerabilities as discussed in Section 2.1.1. The IU property could however require practice to label vulnerabilities with confidence. In addition, the other criteria GEN satisfies are helpful for repeatability. For example, repeatability is impossible without mutual exclusiveness because alternatively, different people would get a choice between different classes that both apply. We argue this criterion should be tested in a real-world scenario. However, evaluating GEN in a real-world setting remains future work and thus so does this criterion.

Criterion number six states that the *Terms are well-defined (6)*, meaning it is industry-congruent and descriptive (see *(3)*) and does not require interpretation. Whether GEN requires interpretation depends on how clear-cut its abstraction layers definitions are. As long as abstraction layers are well-defined, any bug or vulnerability will cover a clear set of abstraction layers. Interpretation *could* pose an issue for concluding "definedness" in exceptional scenarios, but this will be discussed in the Discussion & Limitations section (Section 4.8).

The remaining criteria are already (partly) covered. Criterion seven states that the taxonomy needs to be *Unambiguous (7)*. Concretely, the authors state that the taxonomy should be broad enough (see 2) and should be in an appropriate hierarchy (see 1).

Criterion eight states that GEN needs to be *useful (8)*, in the context that it should not contain a "catch-all" category and only captures "real vulnerabilities". GEN does not contain any residual class (and hence no "catch-all"), and excludes pseudo-vulnerabilities too, as shown in Section 4.4.

According to the next criterion the taxonomy requires *versatility (9)*, referring to clear processes when a novel vulnerability cannot be classified (see 2) and clear processes for keeping up-to-date with current trends and technology. For what (2) does not cover in the second sub-criterion, our example of machine-learning attacks (Section 4.2.1) shows that recent scenarios can be captured — albeit more effort can be required at times.

Finally, the last criterion is concerned with *a representation of human factors (10)*. Here the HCI abstraction layer helps provide the necessary classification options, even though differentiation between different HCI-type vulnerabilities is limited.

Summary. The main issues we found when evaluating the GEN taxonomy have to do with *repeatability (5)* and *industry-congruence (3)&(6)*. Repeatability is mainly an issue because of the resources needed to test it on a large enough scale. Similarly, industry-congruence remains to be seen: we can only hope it will become industry-congruent once businesses understand the benefits. If we consider GEN to not be industry-congruent and lacking repeatability, it would satisfy 13/15 subcriteria and thus 7/10 criteria, considering industry-congruence appears twice within the criteria. Industry-congruence and repeatability is yet to be tested in practice.

4.6.2 AFL

In this part of our evaluation, we evaluate GEN's ability to associate bug-finding techniques with specific GEN classes. American Fuzzy Lop [207], better known by its acronym AFL, is a widely used open-source fuzzer. Because AFL triggers upon *crashes and hangs*, we hypothesize that AFL uses an implicit oracle, mainly finding bugs that fall into the DE-U class. To test this, we perform the first two classification steps of GEN on a list of bugs found by AFL, skipping the third step (vulnerability assessment). To compare, we also provided the CWE class if available, or classified it through CWE if not.

Setup We base our evaluation on a dataset of bug reports gathered from the website of AFL's author, Michal Zalewski. The website refers to a large number of webpages, each describing one or more bugs found through AFL [207]. The links are diverse and can link to an email conversation, a git repository issue, a blog-post or a CVE description among others. Some links are dead, whereas others can describe a multitude of bugs. In most cases, links refer to a conversation where the input found by AFL causes a crash and is shared with the maintainers of the software project. Dead links and sources where it cannot be reasonably confirmed which bug was found by AFL were omitted.

In the remaining sources, the information given was often not enough for GEN classification. More information needed to be found through inspection of what caused the crash. Contrary to the unusable sources, we did include bugs where such information was not available, as this could skew the results. This resulted in 101 analysed bugs that were found by AFL, including the unclassified bugs. The full results are shown in Appendix 2.

Results. According to our analysis, we found that 84/101 bugs (83.2%) are attributed to DE-U. Out of the remaining bugs, 8/123 (7.9%) were attributed to AD-I without resorting to undefined behaviour in the source layer. Most of these cases were assertion failures that trigger a crash as is defined by the source code. Note that assertion failures can be used as technique to stop undefinedness bugs in the source code as discussed in Section 4.5.2. Furthermore, at least one used N-version programming in combination with AFL, gaining a different testing oracle.

One bug (1.0%) was a double-free, meaning it resulted in undefined behaviour of an external library and hence was labelled EF-U[30], and one bug was an AD-U bug since it was left undefined what would happen overall, leading to undesired behaviour (a hang). The remaining 7 (6.9%) bugs could not be determined by GEN due to lack of detail.

As discussed in Section 4.3.2, the DE-U bugs have an AD-IU label (either incorrectness or undefinedness is possible here). Since this could mean there are multiple bugs, it could be argued that fuzzing finds the AD-IU bug instead. However, the undefinedness bug always gets paired with an incorrectness bug[31]. This is one-

[30]A double-free is technically a double bug in the "fixing bugs" format of Section 4.3.2, since it is undefined in the DE-U translation as well. To not skew the results in favour of our hypothesis, it has been labelled EF-U in this case.

[31]Unless incorrectness occurs in the informal layer or the higher layer contains another undefinedness bug.

directional, meaning the incorrectness bug does not get paired with an undefinedness bug by default. Due to the amount of undefinedness bugs and the nature of crashes, this is however unlikely. Furthermore, this would mean that the testing oracle of the AFL fuzzer spans also the informal description layer where it has no knowledge nor any artefacts to find such bugs. In summary, we deem it reasonable to say that AFL mainly finds `DE`-U bugs.

The CWE classification shows that CWE does not hold similar results. The list contained 28 different CWE classes, each averaging 3.6 bugs with a median of 2 bugs. Most common (with 14 occurrences each) were CWE-119 and CWE-125, representing "Improper Restriction of Operations within the Bounds of a Memory Buffer" and "Out-of-bounds Read" respectively. Notably, the list contained 8 bugs classified with CWE-20: "Improper Input Validation" — a catch-all class for the bugs found. Note that this is the basis for all bugs according to the Langsec community. Where CWE struggles in ambiguity and fine-grainedness, `GEN` provides a natural class to bugs found by AFL.

4.7 NGME Vulnerabilities

In this chapter, we tackled the full breath of application vulnerabilities. Not all vulnerabilities can be used to write Next Generation Memory Vulnerabilities (NGMEs) as our work reflects. To start, NGMEs abuse the application memory, which is not explicit until the translation to layer `E`. Since impact can go up but not down (Section 4.5.1), any translation higher than the `DE` (including the HCI and the policy layer) is not relevant in an NGME context. Concretely, we are interested in layers `D` (source), `E` (machine), `F` (process) and `G` (hardware).

Let us take a few examples of vulnerabilities in the above classes. A recurring type is the buffer overflow (`DE`-U) that grants some form of access to an attacker. These are definitely vulnerability types that can be used to write an NGME. `DE`-I vulnerabilities on the other hand (i.e., compiler bugs) are often more niche but could also expose interesting vulnerabilities. The other way around does not need to be the case. For example, a logic bug in the compiler can expose a vulnerability without violating temporal or spatial memory safety principles.

Notably, the source code language needs to be memory unsafe by design (e.g., C, C++, unsafe Rust) to expose a memory violation on the `DE`-U class. The same holds for the Process layer in the external components used on the `EF`-U class. From Process to the machine layer, virtual memory (and even physical memory) is explicit so bugs from either type could expose a memory violation. For example, rowhammer attacks (that fall under the `FG`-I class) change hardware memory and thus can chain into an NGME. If a vulnerability is reported that does not fall in the above classes or the source language restriction, it cannot be used to write an NGME.

4.8 Discussion

`GEN` presents a new way of looking at bugs and vulnerabilities. This hinges on the fact that a bug is a deviation from the intended behaviour. This consequently means that we require some form of ultimate ground truth, which in `GEN` is the informal description layer. We cannot identify any issues with the informal description layer, unless it contradicts either itself or the security standard. Applications that are impossible in practice (e.g., create an application that concludes whether a given program will halt) or descriptions that inadvertently result in unwanted behaviour (e.g., a forkbomb) cannot be captured in this approach.

This issue can become more subtle. As an example, we can build a smart kettle with built-in thermometer at the bottom. The kettle would automatically turn off when the thermometer reads 100°C exactly as the informal description says. However, the user could live high in the mountains where water has a lower boiling temperature. In this case the kettle would keep boiling the water until all is evaporated, since height affects the boiling temperature of water. This issue in the informal description does not show in our model, but presents a fire hazard (and could be called a vulnerability).

This is different from issues due to ambiguity. For example, an informal description could state that we want a working calculator. One can now question whether an operation like `5 + ten` would be defined or undefined. When the informal description (or any layer for that matter) can become ambiguous in some way, the nature of the vulnerability could become ambiguous too. While this could be labelled as an issue of ambiguity, we suggest to always label this undefinedness unless explicitly and clearly specified (e.g., what representation of numbers is allowed).

Abstraction Layers are a source of discussion. Whereas the regular `GEN` layers may seem arbitrary, they do provide us with the full breadth of abstraction layers: from the most informal and abstract representation of the application to a concrete and observable execution. On top of this, we added 2 distinct layers representing the human interaction and a potential overlaying policy the application needs to adhere to. Through this we retain the property of completeness: all bugs and vulnerabilities inside an application can be labelled by `GEN`. The layers in between are designed to map onto course-grained changes in line with bug-finding techniques, as explained throughout this work. While the layer definitions can be versatile, the current layers of `GEN` are carefully designed for its purpose.

We are aware that most individual software-based projects and deployments follow a much more fine-grained procedure with additional abstraction layers that `GEN` omits. A good example would be to add a virtualisation layer or emulation layer when appropriate, as they can introduce a completely new group of vulnerability types. `GEN` could be applied with a custom abstraction layer tree that can become more complex based on the use-case. As an example, the OpenCPI project [145] considers platform, component and application specifications separately. Then it can split the source language layer into multiple parts according to the language used (RCC, HDL, etc.). Alternatively, OpenCPI "components" can be handled separately to individually consider bugs within components, and handling the application layer together with the component specifications where the implementation is a black box (layer `F` in `GEN`). However, these are specialised environments, which was not the aim when designing `GEN`.

Notably, we would have a layer in between the informal description and the formal specification called the "specification" level. Through a layer like this, we can distinguish between the calculator above and a full RFC-like document that presents much more detail (and less room for interpretation). Sadly the difference between the two is hard to define, and it is unclear where to draw the line between whether a given document would be detailed enough to become a specification over the informal description. This has led us to decide that documents describing the applications all fall within the informal description layer.

From GEN, it might seem preferable to keep the number of abstraction layers to a minimum, since fewer abstraction layers make for fewer translations that could contain discrepancies. This is unfortunately an illusion, since the set of translations must all move from one side of the spectrum (the informal description) to the other (hardware execution). Removing an abstraction layer (in practice) increases complexity when translating between the other abstraction layers, making those translations more error-prone. On the contrary, more layers provide a smoother translation from one abstraction layer to the other, making it easier to identify translation bugs [87].

A bug-finding technique could still span over multiple translations. This suffers from the same complexity issue but would be capable of finding a wider range of bugs. Recent work shows potential, particularly with verification of machine code in the Formal-Machine-I class(es) [18, 198]. Regardless, we believe the best way to search for vulnerabilities at this stage is to take separate classes with an appropriate technique in a divide-and-conquer strategy.

Evaluation. The theoretic taxonomy evaluation is based on a set of criteria originating from an attack taxonomy evaluation and not intended for a vulnerability taxonomy. Although the criteria are generic and attack taxonomies often have an "attack vector" or a "vulnerability" vector, better criteria could exist. The taxonomy evaluation does contain subjective criteria (e.g., industry congruence, appropriate hierarchical format) that make it hard to evaluate a given taxonomy. As such, our self-evaluation on these criteria should be read with care when it comes to the subjective criteria. One last criterion was not evaluated due to lack of resources and contacts in industry: repeatability. This is also left as future work.

We categorise the types of vulnerabilities found using one bug-finding technique: fuzzing. We were unable to extend this evaluation to other techniques that are generally not publicly available, such as vulnerabilities found through unit testing. We believe that the nature of the classification gives an intuitive understanding of this hypothesis for other bug-finding techniques. For example, unit tests are used to check a given operation for correct behaviour, i.e., when the source code does not adhere to the supposed behaviour according to the description level. It would be very interesting to confirm this statistically with other vulnerability types, but this is left as future work.

Side-effects. With a side-effect the application does as the specification says, but not solely what the specification says. Side-effects are incorrectness issues in `GEN`. This can lead to confusion as it does not align with the notion of correctness in formal verification. Using Hoare logic [94], the following would be correct:

{L list} sort(L); create_backdoor(); {L sorted}

`GEN` considers this to be incorrect, as they always come at a cost in both performance and additional effects. As such, we do not deem it reasonable to consider applications correct if other unspecified operations are taking place simultaneously.

To illustrate this, consider a pizzeria that recently switched to an online ordering system. Someone orders one pizza, but the system mistakenly creates 5 equal orders for said person. 5 pizzas will be delivered where only one pizza was required (and hence only one will likely be paid for). Although the person ordering a pizza got their pizza, the pizzeria is left with 4 unpaid pizzas. It is easy to see that the online ordering system was incorrect in creating 5 orders.

4.9 Conclusion

In this chapter, we presented a new way of looking at vulnerabilities based on two properties. First, bugs occur when translating between specific abstraction layers, e.g., when translating source code to machine-readable code. Second, the underlying cause of the bug can be explained with the IU property, which states that every bug can be attributed to either incorrectness or undefinedness. The combination of these two fundamental, measurable properties defines different classes of bugs (**RQ2**). Furthermore, we deem a bug a *vulnerability* if it potentially exposes new data or capabilities. We call our classification model `GEN` since it `GEN`erically classifies vulnerabilities. `GEN` satisfies at least 7/10 criteria (13/15 subcriteria) and up to 10/10 taxonomy criteria. When labelling bugs found by the AFL fuzzer, `GEN` classifies 83.2% of bugs into a single class, showing a clear relation between AFL bug-finding and the `GEN` classes. Apart from the inherent uncertainty introduced by the exploitability property , `GEN` provides a tangible and clear definition of a vulnerability with additional explanation on what went wrong.

A body of concepts and folk theorems exists in the community of exploitation practitioners; unfortunately, these concepts are rarely written down or made sufficiently precise for people outside of this community to benefit from them.

Thomas Dullien

5 Taking Control: Hack the Heap

With the uprise of CFI as discussed throughout this dissertation, the attack surface for memory vulnerabilities narrows down to NGMEs. Thus, it becomes more important to leverage the residual attack surface as effectively as possible. One common way of doing this is by overwriting data-structures through heap vulnerabilities.

When faced with a heap vulnerability, a key step for an attacker is to overwrite a useful data structure, e.g., a function pointer, data pointer or an `authenticated` flag (to bypass authentication), to enhance control over the application. Changing the heap layout so the vulnerability overwrites a *chosen* data structure of choice is known as the Heap Layout Manipulation (HLM) problem.

The HLM problem poses several challenges. First, the memory layout for each heap manager is complex, and any heap interaction can completely change the placement of subsequent memory requests. For example, allocating a chunk of memory and immediately freeing it can affect the location of the next allocation. Second, implementation differences between different heap managers can result in different heap layouts for the same application. For example, the commonly used Linux heap manager PTMalloc2, Windows' heap manager LFH and TCMalloc (used by e.g., Google Chrome) work very differently internally. Thoroughly understanding the heap managers' behaviour is therefore vital for HLM, but also extremely time-consuming and error-prone. Third, the heap does not have a generic target to overwrite apart from the heap metadata, and metadata targets can be easily protected [141, 71]. Hence, we need to find memory chunks with data we want to overwrite, for example with the approach of Roney et al. [157].

In automatic solutions [92, 93, 203], a fourth challenge arises. More often than not, the HLM problem is a single step in the bigger picture of creating the exploit.

Thus, not only does the HLM problem need a solution, it needs an *explainable* solution, so the exploit writer is not left with an opaque solution that needs to be reverse engineered before they can proceed.

In this chapter, we draw parallels between the HLM problem and traditional puzzles. From our perspective, the HLM problem shows many similarities with classic puzzles, where solving a puzzle requires logical reasoning and thinking ahead. Puzzle games have been used to successfully crowd-source solutions to complex scientific problems in other domains [199, 200, 44]. Key to the success is to make the game *challenging and fun* while retaining the original goal of creating scientifically meaningful results. Besides, the visual nature of puzzle games often makes the solutions they produce easy to understand and interpret.

We propose "Hack the Heap": an online puzzle game that can represent both synthetic and real-world HLM problems in a visual puzzle format. Hack the Heap (HTH) can be played by anyone regardless of knowledge about the heap, memory or any background in computing or computer science. HTH contains an extensive tutorial system to teach a player how heaps work interactively. HTH does not introduce any computer science terminology or other irrelevant complexity, so anyone can play. In this way, we teach new players intuitively how HLM works and what constraints limit the exploit writer in practice with synthetic puzzles. At the final level of HTH, players can solve puzzles generated from real-world programs, empowering them to successfully perform the crowdsourced tasks by gathering solutions over time. Once solved, the exploit writer can replay the solution(s) in the game interface to see how each solution works and choose the solution that best fits the conditions for exploit writing.

Attacker Model

Without loss of generalisation, we assume that the application under attack is hardened with Write-xor-eXecute (Section 2.5) and CFI. Furthermore, we assume that a vulnerability exists within the application that led to an invalidation of the full memory safety property (Section 2.2.2), specifically on the heap. Note that the combination of the above means that the exploit will become an NGME. We also assume that the attacker can reasonably either create a predictable heap layout (e.g., by causing a crash leading to a restart) or has the ability to learn the layout (e.g., through an ARP). The attacker can interact with the heap by sequencing different operations, where the goal is to overwrite a target of choice: a data-pointer for example. Finally, we assume any non-deterministic behaviour by either the heap manager or the application does not lead to a non-replayable solution. As an example, the Windows heap manager LFH is non-deterministic by nature but can be abused to become deterministic [193, 159]. All of the above are either implicitly or explicitly in line with previous work [203, 93, 92]

5.1 Hack the Heap: The Game

In order to solve the HLM problem, we propose an online puzzle game for anyone to play, where the puzzle board is a one-dimensional jigsaw puzzle representing the heap. Considering that HLM is only one aspect of writing an exploit (or even one aspect of the control phase), it needs to work on behalf of the exploit writer, explaining the solution. The user requires a level of control to make sure that (1) the operations performed in the solution and their ordering make sense; (2) the resulting state of the application (and of the heap) is clear to the user; and (3) the state of the application after the HLM problem is solved is useful for continuing the exploit.

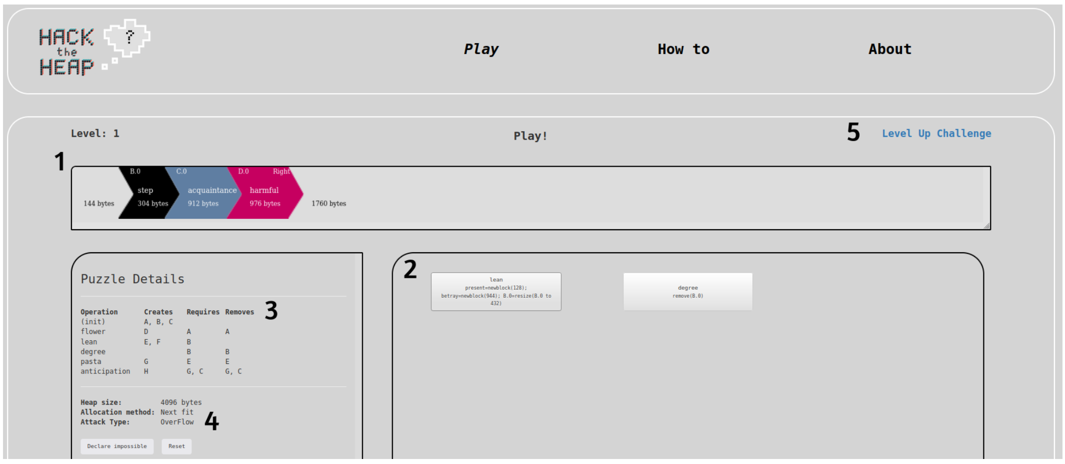
In short, we formulate the following requirements:

R1. The approach should work on a wide range of applications, regardless of application type, complexity or availability of source code.

R2. The approach should not be limited to a specific type (or types) of heap manager.

R3. The approach should not fundamentally discard parts of the solution space to simplify the problem without the ability to return to the original solution space if a solution is not found.

R4. The approach should provide insight into the HLM solution(s), providing all the necessary tools to the exploit writer to continue writing the exploit.



Puzzle Details

Operation	Creates	Requires	Removes
(init)	A, B, C		
flower	D	A	A
lean	E, F	B	
degree		B	B
pasta	G	E	E
anticipation	H	G, C	G, C

```
Heap size:         4096 bytes
Allocation method: Next fit
Attack Type:       OverFlow
```

```
lean
present=newblock(128);
betray=newblock(944); B.0=resize(B.0 to
432)
```

```
degree
remove(B.0)
```

Figure 5.1: A screenshot of the heap puzzle game being played. The numbers correspond as follows: 1. is the playing board (or heap memory space if you like). 2. shows all operations that can be performed at the current stage of the game. 3. shows all requirements of operations before said operation can be used. 4. shows additional details about the puzzles. 5. is a link to the tutorial where the player can level up.

5.1.1 Design

The puzzle board as shown in Figure 5.1-1 will be filled with "puzzle pieces", equivalent to chunks allocated on the heap through a call to `malloc()`. In order to place the puzzle pieces on the puzzle area, a player performs *operations* by clicking buttons, as shown in Figure 5.1-2. These operations represent operations in real-world programs, such as invoking an API call. Operations consist of one or more calls to functions of the malloc-family (such as `malloc`, `calloc`, `realloc`, `free`, etc.).

The goal in the game is to place the bugged and target puzzle pieces next to each other in the game, representing a heap overflow (or underflow) into useful data. To add to the intuition of the game, we simply call these pieces the "left" piece and the "right" piece instead of bugged and target pieces. In Figure 5.1, the last piece (called "harmful") is a *right* puzzle piece, representing a *target* in the HLM problem. If this puzzle piece ends up on the right, adjacent from the piece with the *left* marker, the puzzle is solved.

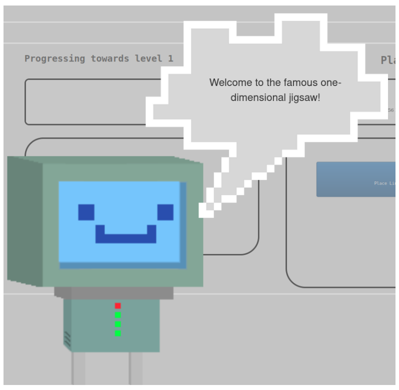

Progressing towards level 1
Welcome to the famous one-dimensional jigsaw!

Figure 5.2: This is the HTH Mascot that guides the player in the tutorial. It also shows up to congratulate the player when successfully solving a puzzle, and when the user thinks that the puzzle is impossible to solve.

5.1.2 Levelling up

To dive straight in the game can be hard for people without intrinsic knowledge about heap internals. As such, we built a level-based system with tutorials to aid people in playing the game and progressing. Our mascot explains the game with the most basic primitives to be able to play the game (see Figure 5.2). For example, only a basic overflow without additional requirements will be presented. Furthermore, puzzle piece placement is done with “first fit” logic, meaning that every next piece will be placed on the left-most fitting position. At the end of this tutorial, the player will be presented with a challenge. Solving the challenge will progress the player to *level 1* as shown on the left-hand side of Figure 5.1.

At this point, the player can play various level 1 puzzles. If the player wants a new challenge, they can press the “level up” button (as seen in Figure 5.1-5). This starts a new tutorial about a new concept (e.g., a new attack type), followed by a new challenge using this new concept. Solving the challenge will again progress the player towards the next level, where they can play puzzles that could include the new concept. We chose this design to let the player decide when it is time to progress further, as different players prefer different speeds before feeling confident within the full extent of each level. In practice, the real challenge is introduced after each level up. Specifically, when playing the regular challenges that include the new concept. The level-up challenge functions as minimal check on the skill of the

Level	Concept introduced
Level 1	Basics (First Fit, Overflow)
Level 2	realloc() action added
Level 3	initialisation function added
Level 4	Next fit
Level 5	Best fit
Level 6	memalign() action added
Level 7	Overflow upon freeing
Level 8	List fit (free lists)
Level 9	Overflow upon allocating
Level 10	Direct pipe to real-world Service fits (e.g., ptmalloc2, jemalloc)

Table 5.1: Different levels in HTH with concepts introduced at each level

player, and as stimulus and excitement trigger to the player. In other words, players can decide their own pace. What type of puzzle is played is shown on the left of the players' screen (Figure 5.1-4). The concepts introduced at each level are shown in Table 5.1. We discussed the educational design in more detail in Section 2.6.1.

Upon hitting level 10, a PTMalloc2 fit will become available that does not simulate its behaviour. Instead, the backend will perform every requested operation using PTMalloc2 and record its behaviour. The results are fed back into the puzzle game so we can ensure a correct representation of the problem. We currently support *PTMalloc2, DLMalloc, TCMalloc and JEMalloc*. This is where the approach switches from educational to a crowd-sourcing platform: we empowered the player to perform HLM and start presenting real-world HLM problems to them.

5.1.3 Visualisation Challenges

The puzzle game shows heap chunks as arrow-like puzzle pieces in different colours. Multiple issues arose in the design of this memory bar. First, we wanted the size of each puzzle piece to reflect its actual size (in bytes). The initial design scaled the different puzzle pieces to their actual size, but this proved unhelpful. Allocation sizes in real-world scenarios stretch over a wide range, creating excessively large chunks. Smaller chunks also became too small to fit all the information (name, amount of bytes, identifier, left/right marker). In some scenarios, puzzle pieces became too small to fit in the arrow-like structure, completely breaking the visualisation. This was solved by setting a minimum size of a puzzle piece (the "step" piece in Figure 5.1). In our current design, the size of a puzzle piece grows logarithmically, so a 1024 byte piece is 4 times as large as a 1 byte piece. With this solution, we cannot scale all the puzzle pieces anymore to fit in the memory bar. Thus, when the bar fills up completely, a second bar is created on the fly underneath. The player can either decide to scroll inside the memory bar or drag the bar down so multiple bars are shown at the same time. This is especially useful when analysing larger applications.

A second challenge is that operations can appear and disappear on the fly, even when performing seemingly unrelated operations. Consider a simple model with three operations: `x=malloc(128);` `realloc(x,512);` and `free(x)`. Upon performing the first operation, new instances of the realloc operation and the free operations will appear. Yet, when freeing the heap chunk, the realloc operation is not valid anymore. These conditional operations are shown on the left-hand side as puzzle details (Figure 5.1, label 3). Puzzle details include which operations create new types, what each operation requires to be executed, and what it may remove (potentially rendering other operations invalid). When searching for other operations to perform, players can always consult this to see what they can do to gain new operations and/or not to lose operations.

Finally, users experienced issues when operations caused changes to the heap they did not expect. In particular, it took time and frustration to find what new puzzle pieces had appeared where, and as a result they often got lost in the new scenario. Hence, we decided to highlight all new puzzle pieces after an operation is performed. This helped the players track their actions while playing the game, making the game significantly more intuitive. Do note here that although the representation might visually be not as accurate as possible, in no way it affects the correctness of the problem space, nor does it affect the solution space.

5.2 Generating Puzzles

In order to play the game, we need puzzles. The HTH game consists of both artificially created puzzles and real-world (recorded) puzzles. Both are represented in the same format when playing the game.

This format as follows:

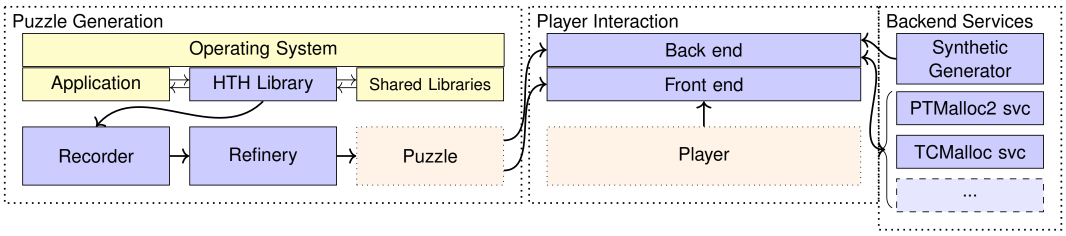


Figure 5.3: The full infrastructure, split in three boxes. The left box describes the generation of real-world puzzles. The middle box describes the user interaction with the website, playing the HTH game. The right box describes the additional services running on the back end of the webserver. The blue colour denotes aspects we developed, whilst yellow shows systems untouched.

[magic] [Fit] [Attack] [heapsize]T [operations]

For example, using the first fit technique (F), attack with an overflow on allocating the overflowing object (OFA) on a heap of size 8096 bytes, we get the following header:

HPM2/FOFA8096T...

An operation is represented with a name followed by a colon, from where every action is listed. Operations are delimited with a period to denote the start of the next operation. Initialisation operations (that are invoked at the start and cannot be invoked by the user) have an additional period in front of its name.

Each *action* starts with a number representing the type of action (see Table 5.2, followed by a *tag* that represents its flow. The tag tells for example what memory chunk needs to be freed. Finally, we add a name with a colon where required, followed by its arguments, in between brackets. As an example, an operation with name “Operation” that allocates 128 bytes and resizes it to 512 bytes afterwards looks as follows:

Operation: 1A(newchunk:128,1) 3A(:512)

The game system automatically recognises which operations can be executed when and what chunk needs to be altered. The context-free grammar of the puzzle format is available in Appendix C.

Number	Action	Args	Name
0	malloc	1	Yes
1	calloc	2	Yes
2	memalign	2	Yes
3	realloc	1	Optional
4	free	0	No
5	mallopt	2	No

Table 5.2: Actions used by the puzzle format. Realloc naming is optional and renames the memory chunk.

5.2.1 Artificial Puzzles

Most puzzles that represent a real-world problem are relatively complex and hard for new players to get into. For example, most real-world puzzles only make sense using a real-world heap manager, but a real-world manager only gets introduced at level 10 (see Table 5.1). For educational purposes, we present easier synthetic puzzles until then. We generate puzzles for the different levels according to Table 5.1. Puzzles are generated based on a range of options (e.g., what fitting technique will it use) and weights (e.g., creating more `malloc`s over `memalign`s).

Yet, this is unfortunately not enough to generate interesting puzzles. For example, if the "bugged" piece(s) and the "target" piece(s) only appear in the initialisation function, then there is no point in playing. Similarly, if the total size of the heap is too small (or too large to perform heap spraying), it can render the puzzle impossible or become increasingly frustrating respectively. On the other hand, if a target allocation directly follows a bugged allocation, its solution is trivial and does not challenge the player. All these corner cases are checked when creating puzzles to ensure the best quality synthetic puzzles and keep the player engaged with the game and the learning process, until players are ready to be challenged with real-world scenarios.

5.2.2 Real-world Puzzles

Ultimately, the HTH game has to be used on real-world applications, as an HLM solution to a synthetic puzzle bears no significance. In order to play real-world HLM problems, we developed the HTH recording framework: a UNIX-based framework that can record heap usage of real-world unmodified applications using any heap manager to turn it into a puzzle. This is done in two steps, as shown on the left-hand side of Figure 5.3. First, we record all calls to the heap manager and save this as a raw trace. This trace is then given to the Refinery, which turns it into a playable puzzle.

Recording Heap Usage. In order to record all heap usage, we preload the HTH library. This library implements wrappers for all functions from the malloc family, to not alter the original program. Instead, it is preloaded with the `LD_PRELOAD` environment variable to make sure our wrapper-functions get called. The wrappers collect function arguments, return value, return address and the pid among others. This information is sent through a message queue to the HTH Recorder. The HTH library contains additional *optional* functions that can be used when linking the application to the library (i.e., if the source code is available). These functions can mark particular mallocs with a custom name or type (bugged/target) statically, where this is marked in the recorder for later use. Alternatively, we offer this information manually, either by passing additional information to the Recorder interface or by manually altering the output later. On the first invocation of any of the overridden (or additional) functions, the library also attempts to locate the main function address. This is sent alongside the `PROCESS_START` keyword to mark a new execution, to calculate the ASLR offset. For the same reason, any dynamic library used (that invokes our wrapper library) will be sent to the HTH Recorder. Together, this can be used later to leverage DWARF debug information (if available) to automatically determine the original variable name as used in the source code.

The information is captured and saved by the Recorder application. The Recorder can be controlled through a CLI, to run the target application, name and mark the start of various operations, interact with the target application through `stdin` and so on. This can be further automated with configuration files. Listing 5.1 shows an example Recorder output file.

The Refinery. We first need to split the recording based on the application's process ids/pids. For multiprocessed applications, any allocation before forking is present in both pids. After each fork, the heap layout will diverge depending on the pid we trace. As an example, Listing 5.1 line 9 shows a fork into a new pid: 81649. The subsequent call to `malloc(4096)` on line 10 is executed on this new pid. Similarly, the calls to `realloc` (line 12) and `free` (line 14) are not present in

```
1   SET target ./application
2   START
3   NEXT init
4   PROCESS_START
5     DYNAMIC /lib/x86_64−linux−gnu/libc.so.6
6       @ 0x7fe454ad1000 @ 0
7     MALLOC(128) @ 0x4011a9 = 0x99e2a0 @ 816848
8     MALLOC(1024) @ 0x7fe454b55e84 = 0x99e330 @ 816848
9     FORK 816848 −> 816849 @ 816848
10    MALLOC(4096) @ 0x7fe454b55e84 = 0x99e740 @ 816849
11  NEXT interact
12    REALLOC(0x99e2a0, 512) @ 0x401220 = 0x99f750
13      @ 816848
14    FREE(0x99f750) @ 816848
15  END
```

Listing 5.1: Recorder Output

this child process. In the parent process trace, we do not want this `malloc(4096)` to appear. Splitting the file leaves us with one output file per pid, from where we can decide which to process.

These single-thread files can then be processed by the Refinery. In short, the Refinery parses the raw data into lists of heap actions per operation. In the parent process of Listing 5.1, this would be an `init` operation containing everything up to the `FORK` command, and an `interact` operation as denoted on line 11 (by the `NEXT` operation keyword) containing the `realloc` and the `free`. Afterwards, it traces the path each allocation makes (by tracing the return values) and labels each path with a unique label. The label tracks which `free`s and `realloc`s correspond with which earlier operations. This trace is colour-coded in Listing 5.1. Here we have three heap traces — yellow, pink and green — where the green one shows a longer trace of a `malloc`, a `realloc` and finally a `free`. In turn, it shows the dependencies, as every `free` requires the chunk to be allocated. After performing the `interact` operation, we cannot perform this again as the green trace has ended. Once we have the operations and labels, we generate a puzzle that can be played on the website.

Some real-world puzzles perform a very large amount of heap-related operations before the user interacts with the application, dubbed the initialisation operation. This is referred to by Heelan et al. [92] as noise: heap allocations that we cannot interact with throughout the course of heap manipulation. In some instances, it adds a burden to the person performing heap layout manipulation (i.e., adding difficulty to the puzzle). In other scenarios however, it adds "noise" to the visualisation and processing only, making it confusing to the player and adding significant loading time. To minimise this, the Refinery can remove some of these actions. In order to remain realistic and conservative, it only removes the middle allocation of three (non-bugged and non-target) allocations *if all three are not freed throughout the execution*. This simplification technique is optional and henceforth refer to this as the "simplified" puzzle.

5.3 Implementation

The game runs on a small Apache2 webserver, available to anyone at `https://hacktheheap.io/`. Besides the webserver, the game uses a database for saving (generated and submitted) puzzles and results. The puzzle game is implemented in TypeScript ES2015 and consists of approx. 5 KLoC. The layout is made with HTML5, CSS3 with Bootstrap and flexbox. The backend of the puzzle game is written in about 1 KLoC Python3 and C. The Heap recorder, library and refinery are written in C and Python3, consisting of approximately 2 KLoC.

5.4 Evaluation

The goal of our evaluation is to show that we can represent and solve real-world HLM problems in the HTH puzzle. We evaluate the HTH infrastructure by generating puzzles from vulnerable applications. Afterwards, we attempt to solve the puzzles, solving the HLM problem for each heap vulnerability. The solution to the puzzles are applied to the original application to confirm it is a real solution to the HLM problem. All tests are performed on a 64-bit Ubuntu 18.04 VM with 4 cores and 8 GB of RAM, using the HacktheHeap.io website.

We first discuss a synthetic example application, taken from a Capture the Flag (CTF) challenge. Afterwards, we discuss the real-world application of Hack the Heap through 3 CVEs in NJS, the NGINX webserver's backend Javascript module.

```
30 typedef struct {
31   uint16_t id;
32   uint16_t content_len;
33   char *content;
34 } blob_s;
35 [..]
36 fread(&(new_obj->id), sizeof(uint16_t), 1, fp);
37 fread(&(new_obj->content_len), sizeof(uint16_t)
38     , 1, fp);
39 new_obj->content = malloc((uint16_t)
40     (2*new_obj->content_len));
41 fread(new_obj->content, 1, new_obj->content_len,
42     fp);
43 fclose(fp);
44 new_obj->content[new_obj->content_len] = '\0';
```

Listing 5.2: Code snippet of the CTF challenge with an integer overflow on line 39–40.

5.4.1 Case Study: Synthetic Example

In our first case study, we use a synthetic example from a non-public CTF challenge. The application provides us with a small interactive shell that can save strings on a given index. In this shell, we can perform various operations such as create a new object on a given id, edit the string, save the object to a file or load from a file. The developer arguably tried to avoid buffer overflows while loading an object by `malloc`ing exactly double the given content-length, creating an integer overflow. The trimmed code is shown in Listing 5.2, showing this vulnerability on lines 39–40.

When a content length is set over half the maximum size of a `uint16_t`, the calculation on line 40 will cause an unsigned integer overflow. Afterwards, lines 41–42 will cause a heap overflow on the `malloc`-ed memory. In puzzle game terminology, this means the puzzle piece created will be a "bugged" piece.

Next up is the question of what to target. Each object consists of an id, the content length and a pointer to the actual content — which is separately allocated. We aim to overwrite the content pointer of a separate object. If we can overwrite this pointer with a pointer value of our choice, we obtain an arbitrary write primitive. This can be used to overwrite the saved instruction pointer, a GOT table entry or to e.g., write a ROP chain onto the stack.

We record the various operations producing a recording similar to Listing 5.1. This is run through the refinery to create a puzzle to be played on the website.

Playing the puzzle shows that we first need to perform a file read operation before allocating the target, then delete the first read before reading the overflowing piece from a file. We can perform this in the original executable to gain the arbitrary write primitive. We finalise the exploit by overwriting the saved instruction pointer to point to a shellcode and gain a shell.

5.4.2 Case Study: NJS

NJS is a limited version of JavaScript, written to extend the functionality of the NginX webserver. In 2019, NJS was fuzzed by various fuzzers including Fluff, a JavaScript fuzzer by Samsung [68]. This lead to multiple CVEs, notably heap overflow CVEs CVE-2019-11839 [53]; CVE-2019-12206 [54]; and CVE-2019-13617 [55][32]. NJS is a particularly interesting case study as the amount of `frees` are limited within the user interactions. All CVEs have been patched.

In preparation for puzzle generation, we marked the allocation functions for objects that start with a pointer as target allocations. We use only allocations of objects that start with either a data pointer or a function pointer, so that an overwrite either creates an arbitrary write primitive or control-flow hijack respectively. Furthermore, the vulnerable version of NJS did not contain any functionality to recognise *what heap action (e.g., mallocs/frees) corresponded with what line* of JavaScript code. We wrote a small JavaScript module (of approx. 50LoC) to send additional messages to the recorder module (see Figure 5.3), marking the start of each individual JavaScript command. Using this, we can see what exact heap actions are performed with each line of JavaScript code. Finally, we marked the 'bugged' allocations for each of the CVEs (in separate runs), i.e., the allocations read/written to outside of its boundaries.

A variety of JavaScript commands were chosen as different operations to create the puzzle, each of which contain one or more calls to `malloc/memalign/free`, to present the player with a set of operations to alter the state of the heap. These always include one or more operations (i.e., lines of JavaScript code) that create bugged or target allocations. Every puzzle is available on the website to be played.

CVE-2019-11839 is a heap overflow on JavaScript arrays. When an array is created, space for an amount of (empty) elements is allocated with an internal `size`

[32]CVE-2019-13617 was found through libfuzzer, the others by Fluff.

variable. If an element is added to the array, it will check if there is still space available. If the allocated space is completely occupied, it will be reallocated into a bigger heap chunk. Alternatively, the first empty space will be used to store the new element. The `array.prototype.shift` operation removes an element from the beginning of the array, and is implemented to move the pointer of the start of the array one element forward. It failed however to update the internal `size` variable, as there is one element fewer that will fit inside the array at this point. A number of `shift`s on an array would cause its actual space to become far less than its internal `size` variable. Elements added at the end beyond its actual remaining size would erroneously not trigger a reallocation of the array, but instead cause a heap overflow until the `size` variable is exhausted.

Upon playing the simplified puzzle, we found a solution: First, we create a new array that we fill with elements until a resize takes place (10). Afterwards, we create a small regex object to fill up empty spots in the memory layout, before creating a JavaScript object. Internally, the JavaScript object starts with a data pointer that becomes directly adjacent to our array. A series of `shift`s will now misalign the size and free space as explained above, and subsequent `push`es to the array will overwrite the data pointer. The resulting heap layout is shown in Appendix D.

Performing the solution on the non-simplified puzzle shows that the same heap layout is achieved, confirming that the simplification did not alter the problem space nor its correctness, while purging 1119 allocations. However, context on how the results are achieved is important to an exploit writer, as this example clearly shows — the exploit writer can analyse the solution in the non-simplified puzzle, even when the simplified version is used to solve the puzzle. When writing a JavaScript file with the above solution, the desired heap layout is not achieved, as the JavaScript parser now creates a slightly different tree (with a different initialisation function, as our puzzle game is considered). A knowledgeable exploit writer does however see that this issue is easily solved by filling the gaps left with a few `malloc` calls. Rerunning the HTH Recorder on this new solution shows that the bugged heap chunk is directly followed by the target heap chunk.

CVE-2019-12206 arises when transforming strings to uppercase or lowercase variants. In particular, these functions do not distinguish between the byte size of a UTF-8 encoded string and its string length. For characters with a representation requiring more than one byte in UTF-8 (e.g., “è”), the string length becomes smaller than the byte size of the string. The new heap buffer created for said string (based on the string length) will be too small to fit all the bytes, leading to a heap overwrite. In contrast to the previous puzzle (from CVE-2019-11839), the overflow here occurs *directly after* allocating the 'bugged' memory chunk. In other words, the target chunk already needs to be in place when the bugged chunk is allocated, and cannot be allocated afterwards as it will overwrite the overflow data. This mode is the “Overflow upon allocating”: level 9 in the HTH tutorial.

In this instance, we created an array and filled it until it grew once. Afterwards, we created a second, empty array to fill up the gap left from the resized first array. Next, we placed a target puzzle piece: this time we used a compiled regular expression object as target. The first fields of the compiled regular expression in NJS contains function pointers to malloc and free, to be used when performing a search with the regular expression. Overwriting these fields would give the attacker the power to call any executable section available.

To finalise the HTH solution, we grow the array again, creating a gap in front of the regular expression object. We finally transform a large UTF-8 string (with enough multi-byte characters) into lowercase, to overwrite the `malloc` function pointer into a value of choice. Here too, this solution is repeated in the non-simplified version and confirmed to produce the same results. In the same way, the solution in JavaScript code presents us with the desired heap layout for exploitation. A random search would have taken on average 84 days, see Appendix E.

CVE-2019-13617 performs an overread when the lexer throws an error. Hence, it occurs during error handling upon parsing the JavaScript file (and hence before interpreting the actual code). Using our solution, this means that the bugged allocation occurs before any operation can be performed (and the application halts before that). Although the lexicographic differences can alter the heap layout when this vulnerability is triggered, our solution does not provide the necessary toolset to find an HLM solution. Within the puzzle, no operations are available (because it halts before any of the operations are executed) and the vulnerability is only available in the initial operation, before the player can perform any action at all.

5.5 Discussion & Future Work

In Section 5.1, we specified 4 requirements for our solution.
R1: Our solution works on a wide range of applications. We do assume interactions can be represented as different HTH operations in a reasonable[33]replayable manner. The main bottleneck of HTH is the players' browser and internet speed when puzzles are large.
R2: HTH supports four heap managers, and the HTH game and its recorder design is heap manager agnostic. This does mean that repeatability remains an issue in non-deterministic heap managers, as a solution may need to be repeated many times for the same result.
R3: The Recorder performs only one simplification that does not change the problem space and hence does not lead to any inaccuracies. Even in the case of distrust towards this or potential future simplifications, they are not fundamentally required to find a solution to the HLM problem. Just like within the evaluation, the non-simplified version of the puzzle can be used to replay (and verify) the solution.
R4: Our solution visualises the heap, providing insight into the heap layout. Different players can find different solutions, that show exactly how and why the solution works. The exploit writer can choose whichever solution works best to continue exploit development.

HTH could benefit from automatic solutions. For example, finding targets could be done by the solution as proposed by Roney et al. [157]. Marking the vulnerable section can be automated, for example using vulnerability patches as done by Brumley et al. [21]. Finally, automatically extracting operations can be done through e.g., fuzzing [207, 68] or symbolic execution [203]. We leave the full implementation of this as future work.

User experience. It would be interesting to further evaluate several aspects of the game's user experience. In particular, user understanding of the game; their enjoyment; their ethical understanding (i.e., making sure they understand that playing it is *not* unethical); and repeatability (i.e., the likelihood of players returning). While this has been done informally on multiple occasions throughout the development of HTH, no final user experience evaluation has been performed.

[33]As an example, the run-time parsing changes in the NJS evaulation were not exactly replayable, yet accurate enough to not prove a burden.

Recorder Extensions. Our evaluation shows part of our limitations as CVE-2019-13617 could not be solved with our work. This is because our methodology uses post-startup operations as interactions, which the CVE did not have. It could be possible to create HTH puzzles by having multiple runs of the application with slightly different initial inputs (e.g., parameters or environment). Comparing the different raw output files could allow to gather the heap-related effects and turn them into suitable operations, synthetically building the expected heap interaction sequence.

Game Extensions. The HTH game can be extended in a variety of ways. As mentioned, HTH could benefit from additional heap manager services, running e.g., ZEND or LFH. Further, HTH can be extended with temporal memory attacks (e.g., UaF or use-before-init attacks) to support other types of vulnerabilities, or to add a distance "success" metric from bugged to target piece.

5.6 Conclusion

We presented a solution to the heap layout manipulation problem by posing the problem as a visual puzzle game. Through the actual usage and probing of a given heap manager, we succeeded in creating a heap manager agnostic solution (**RQ3**). The game can be played online through a browser and is described in layman's terms with an extensive tutorial system. We also presented a heap recording infrastructure, where the heap interactions of real-world applications can be recorded. Applications with a spatial heap vulnerability can be recorded and turned into a puzzle, where the solution to the puzzle represents an accurate solution to the heap layout manipulation problem. We showed its real-world usage and limitations through different CVEs, with solutions that remained correct when applying them to the application. The game also serves as a severity assessment: a solved real-world puzzle is a very beneficial scenario to exploit writers, as an overwrite or overread into memory of choice can go a long way in exploit writing.

6 The Payload: System Call Argument Integrity

Data-only attacks are performed when many control phase mitigations are in place limiting the control phase, most significantly CFI. While gaining Turing-Completeness (TC) is theoretically possible [98] with data-only attacks, Arbitrary Code Execution (ACE) remains limited [109]. For exploit writing, gaining ACE in the control phase is generally not necessary — as long as the required payload is executed. In some examples ACE had not been achieved, but the right interaction was found to successfully execute the payload. Notably, the data-only attack in Nullhttpd [32] can spawn a shell and the ProFTPd attack [34] reads out a private key.

Existing solutions against data-only attacks protect the data from tampering but suffer from a large performance overhead. To limit the performance overhead, defences have tailored the data protection towards what they deem important to protect, as discussed at the end of the timeline in Section 2.5.

We believe the crux of protecting against data-only attacks lie within the gateway to functionality: *system calls*. Any application requires system calls to implement their input and output. For exploit writers to impose their behaviour upon a vulnerable application, system calls are required that provide the functionality that the exploit writer wishes. In the context of data-only attacks, functions that implement these system calls need to be called with a "natural" appearance. Availability of these system calls cannot be avoided without severely restricting the allowed functionality of an application [12]. Yet, it is the tampered data in the exploit *that is used as system call argument(s)* which turns the benign and intended behaviour into the payload.

To prevent this, we propose System Call Argument Integrity: a practical mitigation technique for protecting against data-only attacks. System Call Argument Integrity protects the data-flow of exactly those system calls an exploit writer is likely to use. To be specific, it enforces memory segmentation between security-sensitive system calls and the remaining data-flows through Intel MPK [46]. Intel MPK is already available on the market for multiple years and provides fast, thread-specific protection against memory-based attacks. As opposed to existing solutions, only sensitive flows require instrumentation. When MPK-enabled hardware is not available, we present alternate protection schemes using `mprotect` or Software-Fault Isolation (SFI). On top, our solution requires no kernel modifications or instrumentation of shared libraries.

6.1 Threat Model & Requirements

In this chapter, we propose a solution against data-only attacks: attacks that are possible in the case of strong CFI protection. Thus, we assume mitigations against control-flow attacks are in place (i.e., forward CFI and a shadow stack). To be exact, we assume no illegal control-flow occurs. Memory cannot be both writable and executable (NX), as this could break the CFI solution, and the address space layout is fully known to the attacker. While this may be an overestimation of a typical attacker's knowledge, our threat model forces us to protect against sophisticated attacks. This is consistent with previous work [109, 79]. Finally, we assume the attacker has an AWP that allows the attacker to overwrite arbitrary memory locations with a value of its choosing at a certain point in execution.

There are two cases of exploits we do not consider. First, we consider only data-only attacks arising from memory safety violations. If the user input already goes straight into the system call (For example, command injections like Shellshock/CVE-2014-6271 [52]), data-flow integrity will not block this attack. Secondly, we do not consider the semantic importance of variables from within the program. If the program contains an "authenticated" flag or a private key for example, it may be a good attack vector. We consciously refrained from protecting flows with semantic significance, in order to keep this approach fully automatic and readibly deployable. This is reflected in our attacker win condition.

Attacker win condition. In our model, we focus on attackers whose goal or *win condition* is to gain control over the *underlying system* with the user privileges of the application being attacked. Gaining control in this context is a broad goal that includes gaining remote access on the target machine or switching to an account with new privileges. We refer to this win condition as a Privilege Escalation Primitive (PEP). This model therefore excludes data-only attacks that compromise the application itself (e.g., to leak its sensitive data) but do not (directly) seek to control the rest of the system.

6.1.1 Design Requirements

Given our threat model, we next define the requirements for a good solution.

(R1) **Minimal Developer Effort:** To maximise impact, mitigations should be compatible with legacy applications. For legacy applications, we consider solutions requiring any developer effort or annotations beyond recompiling the application to be impractical. Furthermore, any mitigation must not affect the program semantics (i.e., the program should not break under normal use).

(R2) **System Compatibility:** The solution should be easily deployable. We therefore require no kernel modifications or custom hardware extensions. In addition, we aim to avoid modifications to dynamic linked libraries.

Intel MPK places a limitation on system compatibility, but MPK supporting machines are already available and no customisation is necessary beyond this. We believe that a hardware-enforced mechanism is in line with previous protection mechanisms such as the NX bit/DEP [186] and soon CFI [120].

(R3) **Security:** Any proposed solution must significantly improve security with respect to the threat model. This means preventing attacks that are possible with mitigation techniques like CFI in place. Although we trust the kernel, dynamic libraries may be abused by attackers [57, 56], and any solution must therefore protect against vulnerabilities within dynamic libraries.

(R4) **Acceptable Overhead:** A mitigation technique is only viable in practice if its performance overhead is reasonable in a production setting[34].

[34]We purposefully do not mention any concrete numbers here, as this is not an objective measurement. An extreme boundary may be the average DFI overhead of 104% [28].

6.2 System Call Security

Within an application, various different data-flows exist that do not interact with each other. When performing a data-only attack, the AWP connects the source of the write primitive to an arbitrary data-flow where such interference was not intended to exist. In practice, some data-flows are more security-sensitive than others. The goal for data-flow integrity is to block the malicious data-flow interference into a security-sensitive flow [28]. In particular, data can flow into a *system call argument* that is used to perform an operation that is of interest to attackers.

This is why we present System Call Argument Integrity (`SCAI`), a form of data-flow integrity tailored towards system calls. Since any user-space process does not have any in- or output without system calls, the key reasoning is to protect the arguments of system calls against tampering from adversaries. This prevents malicious interaction with the underlying system and thus with the outside world.

6.2.1 Security Sensitive System Calls

We hypothesize that not all system calls are used or even usable by attackers. In order to validate this hypothesis and to determine which, we performed a three-fold analysis. First, we analysed the system call usage of 274 shellcodes. Then we looked at proof-of-concept exploits in published work. Finally, we manually went over all system calls on Linux `x86-64` to exhaustively separate security-sensitive system calls from harmless ones.

Shellcodes. Shellcodes contain generic, transferable payloads and thus fall within our threat model. To understand the usage of system calls in shellcodes, we analysed 239 shellcodes for `x86` and 35 shellcodes for `x86-64` using the shellcode database from `shell-storm.org` [161]. For every shellcode, we counted the different system calls used. When a system call occurs multiple times within the shellcode, this does not influence the total count and is counted a single time. The results are shown in Appendix F.1 for `x86` shellcodes and Appendix F.2 for `x86-64` shellcodes.

Although many system calls are available[35], the tables clearly show a small subset of system calls: ±30 different system calls are being used in at least 2 shellcodes.

[35]382 on the Linux master branch at the time of writing [189]

Moreover, both tables have `execve` as number one occurring system call with a combined *172/274 shellcodes (62.8%)*. This confirms `execve` is widely used for exploitation purposes.

Published work. Within research publications, the tendency towards a small set of system calls is seen too. To show this, we looked at 12 papers, ranging from 2007 to 2019. All of them either (1) contain a new technique for exploit generation or (2) contain a Proof-of-Concept (PoC) exploit as part of a case study. For each of these, we determined if their PEP was made explicit and what PEP was used to finalise exploitation. Out of the 11 papers that specified what PEP they used, 9 mention `execve` or a derivative and 4 papers refer to the use of shellcodes. The complete results are shown in Table 6.1, at the end of the chapter.

Audit. Besides analysing what is currently used in practice, we wanted to build a theoretical boundary on what is potentially useful for attackers. For example, the `execveat` system call is rarely used but provides `execve` functionality — the most used system call in the previous analyses. To ensure we cover all potentially harmful system calls, we manually audited *all* system calls of Linux `x86-64` to perform a system call audit and determine potential security (ab)use.

From the full list of all system calls, we first removed unavailable system calls and left out system calls that have no arguments or impact on the underlying system (e.g., `getpid`). Afterwards, we removed all system calls that need superuser privileges to run, since this is out of scope according to our threat model. This is the case for 60 system calls and can be taken into account when necessary to run a program under root privileges. They are marked separately in the audit in Appendix G.

From the resulting set, we labelled the system calls based on whether they can potentially be used for exploitation. This could be a single system call or a combination of system calls that can be chained to write an exploit (e.g., `open` $\rightarrow$ `write` $\rightarrow$ `close`). If more than one system call is necessary to create an exploit, we label any potential start of a chain (e.g., `open` in the example above) as dangerous. Since an attacker first needs to control a starting system call in a chain, we remove any follow-up system calls. This is consistent with previous work [12]. For any other chained system call, we remove them if they require a different system call argument compromise as part of the attack prior to being called. Finally, we manually inspected the remaining list and remove any item that is certain not to lead to any privilege escalation. The remaining system calls are listed as security-sensitive. Together with the chain start system calls, this analysis provided *32* security-sensitive system calls. This is mostly consistent with earlier work [12], with a small divergence due to a difference in threat model. The full audit can be found in Appendix G.

6.2.2 Sensitiveness Boundaries

The audit represents a theoretical upper bound, whereas the other two analyses represent practice. Hence, we will end up with two sets of system calls that are are considered security-sensitive from different perspectives.

Regardless of the practical analyses, a few of system calls *need* to be taken into account. This is in order to preserve the function of the NX-bit (and with it, CFI): we need to ensure no writable and executable memory chunk can be crafted by an attacker. Creating such a memory chunk requires control over the protection bits of the following system calls: `mmap` (0x9), `mprotect` (0xa) and `pkey_mprotect` (0x149). For any solution to work, attackers must not get access to the protection bits of any of these system calls.

From the shellcodes and the published work, we can conclude that `execve` (0x3b) is clearly key within exploitation in practice. This system call on `x86-64` is semantically equivalent to `execveat` (0x142), `execve32` (0x208) and `execveat32` (0x221). Hence, we can conclude with certainty that protecting these system call arguments is necessary. Together with the NX-bit circumventing system calls in the paragraph above, we find 7 system calls that are crucial. We refer to this set of system calls as *in-the-wild*. The *in-the-wild* system call list is available in Appendix H.1.

On the other hand, our audit bounds the set of system calls that *may* need to be protected with `SCAI`. This is a superset of *in-the-wild* and arguably contains system calls that are irrelevant in the bulk of situations. We refer to this set of system calls as *in-audit*. The *in-audit* system call list is available in Appendix H.2. In the remainder of this work, we consider these two sets in particular, but other sets (based on other threat models) can be used instead.

```
 mov rax, r15 ; pointer
 or rax, 0x10000000 ; mask1
 and rax, 0x10ffffff ; mask2
 cmp r15, rax ; check if in region
 jne success ; if not in region
 shr rax, 0xd ; get the identifier
 cmp rax, 0x2 ; 0x2 is allowed
 je success ; continue
 callq exit@plt ; the check failed
success:
 mov [r15], rax
```

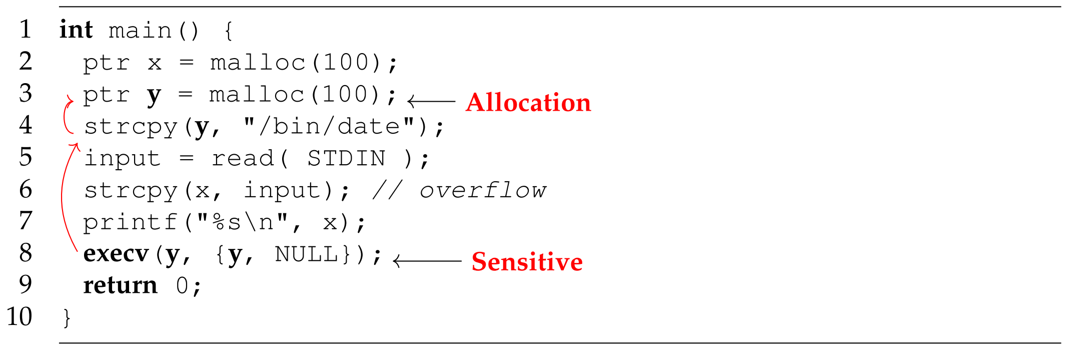
Listing 6.1: Example instrumentation of a write instruction when using SFI and bucketisation

6.3 System Call Argument Integrity

SCAI tracks data-flow from data allocation into a security-sensitive system call. At a high level, we determine if an allocation (stack/heap/globals) has a data-flow into a security-sensitive system call. If so, the allocation is changed and allocates the memory in a predetermined secure memory region. These data-flows are considered security sensitive data-flows, or sensitive data-flows in short. Depending on the memory protection mode, this memory will be set to read-only using the Intel MPK mechanism or an mprotect system call, and will only be set to writable when necessary. Alternatively, we can mask all writes with SFI to enforce the data-flow integrity. In Section 6.2, we identified two sets of security-sensitive system calls: *in-the-wild* and *in-audit*. Protecting either set of system calls with SCAI is referred to as *SCAI-wild* and *SCAI-audit* respectively.

6.3.1 Overview

In this section we give an overview of how SCAI works using a minimal example. Consider the example code in Figure 6.1. Line 6 contains a heap overflow since no buffer size is specified. If the user input contains more than 100 bytes, it overflows into the second buffer created. This overflow provides a contiguous memory write and can be exploited to create a bash shell. For simplicity, we do not turn this vulnerability into an arbitrary write. Given that the two mallocs are allocated



```
 1  int main() {
 2    ptr x = malloc(100);
 3    ptr y = malloc(100);
 4    strcpy(y, "/bin/date");
 5    input = read( STDIN );
 6    strcpy(x, input); // overflow
 7    printf("%s\n", x);
 8    execv(y, {y, NULL});
 9    return 0;
10  }
```

Figure 6.1: Example vulnerable code with a backwards data-flow trace.

adjacent to one another, an attacker can give an input long enough to overwrite the value of `y` and write `/bin/sh` to execute a bash shell instead.

The aim is to block this attack without changing the semantics of the program. In order to do so, we import data from `libc` and recognise a flow from `execv` (line 8) to a sensitive system call (`execve`). `SCAI` tracks the data-flow back to the `strcpy` instance on line 4 and the `malloc` on line 3 as shown in the trace in Figure 6.1.

The `malloc` function is recognised as a memory allocation source so we rewrite this to use a custom allocator in the secure region. Now this memory area is read-only for the duration of the program. Afterwards, we traverse this `malloc` forwards again to find any uses. Every memory write is wrapped with two function calls. Before the instruction `set_safemem_writable` is added to give the write instruction write privileges to the memory segment. After the instruction, we reset this again by calling `set_safemem_readonly`. This is done around the `strcpy` instruction (line 6) and around the `execv` instruction (line 8). The instrumented version is equivalent to the code in Listing 6.2, highlighting the changes. When protecting using the SFI scheme, write instructions are masked instead of a read-only property being enforced. An example mask instrumentation is shown in Listing 6.1.

6.3.2 Compiler Design

In this section we describe the complete `SCAI` framework, explaining in depth how we secure binaries during compilation. We analyse both executables and dynam-

```
int main() {
  init_safemem();
  x = malloc(100);
  y = safe_malloc(100);
  set_safemem_writable();
  strcpy(y, "/bin/date");
  set_safemem_readonly();
  input = read( STDIN );
  strcpy(x, input); // overflow
  printf("%s\n", x);
  set_safemem_writable();
  execv(y, {y, NULL});
  set_safemem_readonly();
  return 0;
}
```

Listing 6.2: Instrumented version of the example in Figure 6.1

ically linked libraries alike. After the analysis, we either export the results in the case of a library, or instrument the code if we are compiling an executable. A library export can subsequently be imported to understand the data-flow and security-sensitivity of used library functions within the application. The full process of 7 steps is depicted in Figure 6.2. We next discuss each individual step in detail.

1. System Call Identification. In order to harden the arguments to the system calls, we need to find the system call invocations. If the system call that is being invoked matches one in our list of security-sensitive system calls, `SCAI` looks up which arguments to this system call are security-sensitive. This is the **system call identification (1)** step. If we cannot determine which system call is invoked, `SCAI` takes *every* argument as well as the system call number itself (the `rax` value in x86-64).

2. Function Dependency Analysis. Concurrently with step (1), we perform a **function data dependency analysis (2)** to determine data-flow dependencies between arguments, the return value and global variables. This lets us analyse functions separately, as well as providing tooling for uninstrumented libraries. This provides us with information showing whether a data-flow was detected and gives a high level overview of certain data-flows from function to function. If external functions are called (i.e., from an imported library), an import from a previous analysis can fill in the dependencies.

3. Minimal Sensitivity Analysis. We trace system call arguments from step (1)

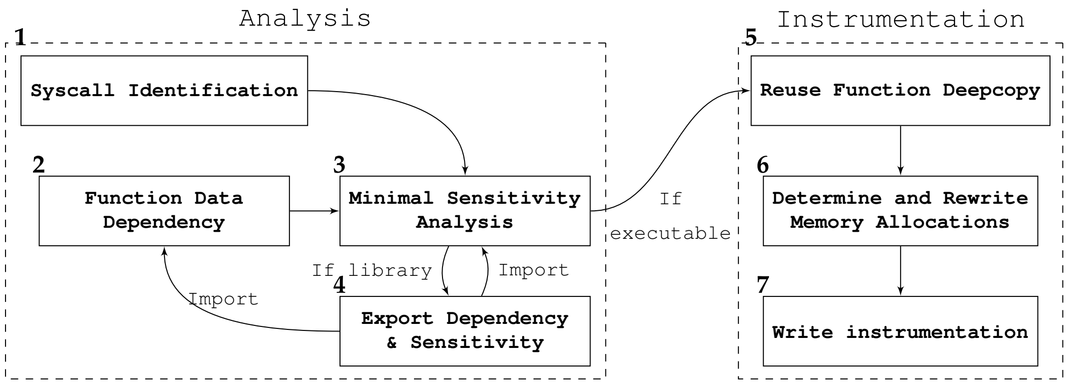


Figure 6.2: SCAI Design

and from imported libraries to their source. While tracing, we likely cross multiple function boundaries. If a function argument flows into a sensitive system call argument, we know this function argument is always sensitive. We refer to the subset of function arguments (and return value) that are always sensitive as the **minimal sensitivity (3)** of the function. This is in contrast with the function calling a separate function (that might be used in different contexts): we discuss this further in step (5) (reuse function deepcopy). After this step, we know the minimal sensitivity of each function, as well as any sources that potentially end up as a system call argument.

4. Export Dependency & Sensitivity *(libraries only).* When compiling a dynamic library, we want to save the analysis results for later use. We **export (4)** the function data-flow dependencies as well as the minimal sensitivity for each of the functions. Source information is dropped for libraries since we will not instrument them. This did not prove limiting in practice, since most source analyses ended in (exported) function boundaries or immutables/constants. This step is where the SCAI procedure ends for code that should remain uninstrumented, i.e., dynamic libraries.

5. Reuse Function Deepcopy *(executables only).* A call *into* a function needs to be instrumented. Unfortunately, the same function may be reused at different callsites. At some callsites, it could be used without requiring privileges, invalidating the "always sensitive" property from step (3) (minimal sensitivity analysis). We still have to instrument (i.e., grant privileges to) these functions for the application not to crash — although not all callsites may need said privileges.

This issue is most easily illustrated with an example. Consider our target program uses a custom `my_memcpy`. If any sensitive data-flow uses `my_memcpy`, it will have to be instrumented to allow it to write to secure memory. At the same time, it is undesirable to have *every* call to `my_memcpy` be privileged: this poses a security issue as well as adding additional overhead. In contrast to the "always sensitive" property from step (3), these functions are "potentially sensitive".

We solve this by creating a **function deepcopy (5)** and instrumenting only the copy. Any call to a function with a security-sensitive data-flow will then use a cloned privileged function rather than the original. This limits the instrumentation as well as improving security: any other call to the original function is left untouched. This needs to be done recursively as well within our newly copied function, hence the *deep*copy.

6. Memory Allocation Rewrite *(executables only).* We are now ready to add the secure memory mechanism to the binary. From step (3) we know what system calls are used and the memory sources, i.e., the only memory allocations that can contain security-sensitive system call arguments. In this step we **rewrite these memory allocations (6)** to use the secure memory.

7. Write Instrumentation *(executables only).* After rewriting the memory allocations, we need to add **write instrumentation (7)** to ensure any expected write instruction into the secure memory allocations will not cause an error. Here, we need to perform an additional forwards data-flow analysis since the memory allocations can be (re)used in locations other than those identified by the initial backwards data-flow analysis performed in steps (1–3). Failing to instrument these writes leads to false positives.

How to instrument a write instruction depends on how the secure memory region is protected. `SCAI` supports three mechanisms (discussed in more detail in Section 6.3.6): Intel MPK, `mprotect` and software fault isolation (SFI)). For MPK and `mprotect`, we wrap write instructions with two functions: one to make the secure region writable, and another to make it read-only again afterwards. Note that for (deep) copied functions, we instrument the copied versions by design while leaving the original functions untouched. For SFI, we calculate a bitmask of the secure region and instrument all remaining memory write instructions instead. At the end of the instrumentation, we add an initialisation function at the start of `main` to initialise the secure memory region.

6.3.3 Bucketisation

`SCAI` relies on attackers not gaining control over any instrumented write or call instructions. When this does happen, basic `SCAI` does not provide any protection. We cannot completely avoid this since the system call functionality is required. Yet, writing an exploit with some security-sensitive system calls (e.g., `open`) may be non-trivial — in contrast to other system calls (e.g., `execve`). To overcome this issue, `SCAI` can split the secure memory in different regions (i.e., *buckets*) for different system calls.

With bucketisation enabled, every privileged write primitive can only write to the parameters of system calls for which there exists a data-flow relation. For example, an attacker that controls a write that is instrumented because of the `open` system call can influence other system calls to `open` but not any arguments to `mmap`. This raises the bar for leveraging complex and non-trivial system call usage in comparison to straightforward system calls when writing exploits. Moreover, adding more system calls to be protected no longer increases the attack surface.

6.3.4 Dynamic Libraries

The analysis described in the previous section starts at the system calls. However, most system calls used nowadays are implemented within a wrapper function, typically in a `libc` or `libcxx` library. Before we can instrument applications and protect their system call arguments, we therefore need to analyse these libraries. Besides, these libraries contain core functionality often used within applications. In other words, it contains crucial data-flow information. Instead of manually analysing and mocking or modelling these functions, we perform the same data-flow analysis as used in the application. As such, we compile dynamic libraries without instrumentation as part of our analysis (*(R2) System Compatibility*) in steps 1–4. Apart from the compiled library, `SCAI` will export a file containing all necessary details to fill the gaps in subsequent analyses of the main application code.

Note that dynamic libraries generally do not change much over the course of time by design, since a public API defines the behaviour of the library. This API generally implies what data flow relation exists through its intended behaviour

(See Chapter 4). In particular, libraries containing system calls typically have a stable export. We exploit this observation by exporting one set of data flows and reusing this export even when versions of libraries change. The uninstrumented, newly compiled libraries do not need to be installed, as the ones currently available most likely have the same data-flows and system calls. Instead, more recent versions of dynamic libraries may add new functionality which only requires recompilation if this functionality is used within the application.

6.3.5 Data-flow Analysis

`SCAI`'s data-flow analysis is performed on an SSA language [158]. For intra-functional analysis, we traverse the explicit UD chains. On inter-functional analysis, we extend this by matching the traced value to its argument or return statements in a context-insensitive manner. For indirect function calls, we perform a matching function prototype analysis. Upon finding an external function, we use the imported information gathered from dynamic libraries to resolve data-dependencies if possible. If this is not available `SCAI` considers any value to be potentially affected, avoiding crashes under normal use.

6.3.6 Memory Protection

To protect the secure memory region, a custom library is used to handle the secure memory requests, set memory writable (and back to read-only) and to initialise the memory region. This custom library is available with three different mechanisms: mprotect, SFI, and MPK. All are implemented using DLMallocs' mspaces and wrapped with x86-64 assembly for MPK opcodes and direct system call invocations. Each protection mechanism has separate benefits and drawbacks which we will discuss below.

mprotect Through the `mprotect` system call we can enforce a read-only property on memory. Since this enforcement is not thread-specific, `SCAI` with `mprotect` protection can fail in a multi-threaded setting. Besides, the system call incurs a large overhead so this method is mainly suited when required protection is limited.

SFI Software-Fault Isolation (SFI) masks pointers with a bitmask to either check if the memory region written to is correct, or to force the pointer inside or outside a given region. In contrast to the `mprotect` solution above, multithreading is not an issue. However, all memory write instructions require instrumentation to block writes into secure memory, simulating the read-only property. In contrast to `mprotect`, SFI cannot protect dynamic libraries unless a recompiled instrumented version is used. Besides, SFI regions do not scale dynamically.[36] Summarising, the SFI overhead does not scale with the data-flows to be protected and cannot protect external library functionality without recompilation, but does support multi-threaded applications.

MPK As the best of both worlds, we can use Intel MPK to enforce the read-only property on secure memory regions. MPK protection works on a per-thread basis, so multi-threaded applications can individually gain access to memory when necessary. MPK also adds the lowest overhead of the three options: only sensitive data-flows need instrumentation and MPK does not need system calls (i.e., kernel interaction/context switch) after initialisation. It does require hardware support, which may not always be available to the user. Nonetheless, it solves all problems at hand and MPK-enabled CPUs have been on the market for several years already [104].

6.4 Implementation

`SCAI` is implemented as an LLVM link-time optimisation (LTO) module pass on LLVM 9.0.0 (commit 69716394f3d) in ±3.9 KLoC of C++ code. This is complemented by a linker script for memory segmentation and a custom dynamic library to provide the memory protection, containing approximately 800 LoC in x86-64 assembly per memory protection technique.

The memory allocated is currently statically determined as a proof-of-concept. This means a fixed amount of 16MiB or 1MiB per bucket is allocated upon running the binary. For mprotect or MPK, the Heap Manager can be customised further to make the secure memory scalable and/or thread-specific.

Exporting and importing the dynamic library data is done with YAML. For extracting the data dependencies and system calls we used musl libc [135]. The secure memory management is implemented on top of dlmalloc [122] using mspaces and pthread mutexes.

[36]This is unless the bitmask is writable during runtime which breaks with our threat model.

6.5 Evaluation

In Section 6.1.1 we defined four requirements for any solution: (R1) Minimal Developer Effort; (R2) System Compatibility; (R3) Security; and (R4) Acceptable Overhead. We discuss all 4 requirements below with respect to `SCAI`.

(R1) **Minimal Developer Effort.** Our implementation performs automatic compile-time instrumentation, but relies on security-sensitivity and data-flow information for any dynamically linked libraries. Analysis results for common dynamic libraries can be generated once and shared to reduce the additional effort required. Custom libraries will still need to be analysed, but this only requires running the compiler pass. Thus, some limited additional effort may be needed. On the other hand, `SCAI` does not require the developer to mark sensitive information (i.e., with attributes): the process of recognising security-sensitive data-flows is automatic. Thus `SCAI` requires no manual effort on the part of the developer beyond compilation flags, and does not require any application-specific knowledge.

(R2) **System Compatibility.** We rely on the Intel MPK mechanism to provide us with memory-based per-thread protection. This limits the deployment of `SCAI` to MPK-enabled computers, and limits the amount of protection domains available for other MPK usage. As seen in the past, Intel provides such hardware support for a small range of CPUs until either interest shrinks (e.g., Intel MPX) or grows (e.g., Intel SGX). A continued interest from the community into Intel MPK technology — together with deployable solutions — can make MPK a valuable addition to be widely deployed in the next generation of CPUs. Alternate solutions are available for `SCAI` using `mprotect` or SFI, each with their own drawbacks.

Besides MPK, our solution works out-of-the-box on any Linux system. For other OSes (e.g., Microsoft Windows) a system call analysis is required similar to Section 6.2.1 to protect the right data-flows.

(R3) **Security.** `SCAI` is clearly effective in our motivating example in Section 6.3.1. To recap the example, an exploit writer can chain the heap overflow into the first argument from `execv` — the argument that decides what external application is executed. `SCAI` disrupts the stitching of the two data-flows, protecting the `execv` call by restricting write-access.

Theoretically, we should also protect against real-world data-only attacks, as long as the data-flow under attack is not already a security-sensitive data-flow. To show this in practice, we analyse Nullhttpd 0.5.0, a webserver application with a known data-only attack. The full analysis is described below.

(R4) **Acceptable Overhead.** Intel MPK stores the protection flags in a separate register that is indirectly available from user-space. After initialisation, we do not need to switch to kernel-space and back (like `mprotect`, conditionally branch (SFI) or write to memory locations (Like most DFI solutions [136, 3, 137]). Through these properties, the overhead is significantly limited.

The main performance overhead is the result of rewriting (fast) stack/global variables into heap allocations when they are security-sensitive. Global variables present a small initialisation slowdown, but creating and removing stack variables is much faster than a heap-based allocation/free. While this is limited to security-sensitive stack variables, it can slow down applications under repeated calls to such functions. We discuss potential solutions for future work in Section 6.6. A practical evaluation on the resulting overhead is below.

6.5.1 Security Evaluation

Assessing the security benefits of applications is generally a difficult task. Only a few data-only exploits under our threat model are publicly available since CFI is not deployed at scale yet. Besides, small changes to the application, compiler version or environment can change the complete course of exploitation. Our security evaluation focuses on the webserver `nullhttpd` 0.5.0 with vulnerability CVE-2002-1496 [49]. The CVE is used by BOPC to generate a data-only exploit that calls `execve` [109], and has a thorough write-up as done by Chen et al. [32] for 32-bit.

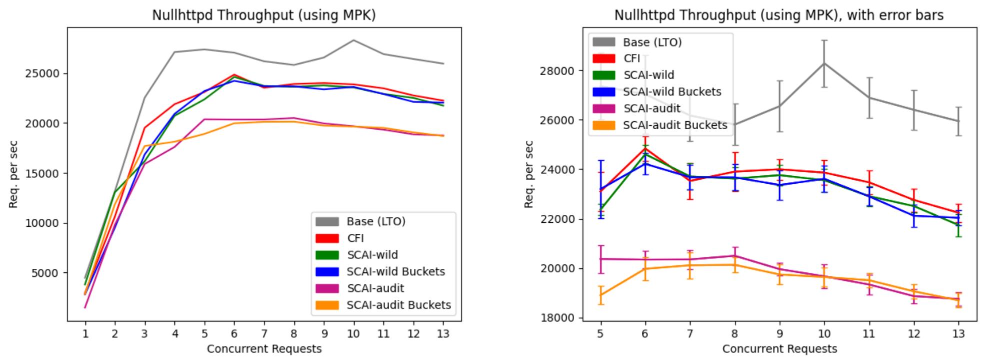
Nullhttpd 0.5.0 contains a signedness error on the content length value in the header of HTTP requests, allocating a small buffer for the POST data. A larger POST data size will overflow in the heap when writing to this new buffer. This heap allocation and write is not instrumented by `SCAI`-wild. Because the POST data could be forwarded to a CGI application, `SCAI`-audit does instrument this allocation through the `open` system call, making it available to the security-sensitive memory. In other words, `SCAI`-audit requires bucketisation to isolate the arguments to `open` from the arguments to `execve`. `SCAI`-wild protects against this with and without bucketisation because `open` is not part of the *in-the-wild* system call set.

The overflowing write is done through a *memcpy* instance. This shows the importance of the Reuse Function Deepcopy (5) step when statically linking, or proper dynamic library handling when linked dynamically. If the *memcpy* function was instrumented (e.g., because loading the configuration file requires an instrumented `memcpy`), the exploit would successfully spawn a shell. Our implementation successfully distinguishes between different uses of functions and blocks the exploit.

Using the `mprotect` mechanism, race conditions occur since Nullhttpd is multithreaded. Nullhttpd works with a dispatcher that receives the request, before spawning a new thread to handle it and listening for the next request on the main thread. It can be configured to run on a single-thread (meaning the dispatcher will wait until the previous request is handled), which helps as long as the dispatcher does not invoke the memory protection functions. In the case of bucketisation, it cannot invoke the same bucket instead: this will introduce race conditions. this occurs in `SCAI`-audit (with and without bucketisation), meaning that the program can break with a false positive under normal conditions. Similarly, if mutexes are used, this would create a deadlock. The dispatcher is not instrumented in the `SCAI`-wild version of Nullhttpd, meaning we just need to eliminate concurrency issues among request handling threads. Through a single request handling thread or protection through mutexes, `SCAI`-wild can work under the `mprotect` mechanism: this protects against the known attacks just like the MPK version. Unfortunately deploying it is unrealistic due to performance issues, as we will discuss next.

6.5.2 Performance Evaluation

For the security evaluation we also used Nullhttpd 0.5.0. Using this webserver, we performed a stress test to see how it would perform with the added protection from `SCAI`. The stress test is performed using WRK2[37]over a period of 1 minute per test with increasing concurrent requests. The machine has an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz with 80 cores and Intel MPK support, and 252GB of RAM, running Ubuntu 20.04.4 LTS (focal). All libraries and executables were build with LLVM 9.0.0 [40]; LLVM safe stack and (forwards) CFI [188] (unless mentioned



(a) Full Nullhttpd 0.5.0 throughput with MPK. (b) MPK throughput starting at 5 concurrent requests, with error bars.

Figure 6.3: Nullhttpd 0.5.0 throughput as a function of concurrency, using Intel MPK as protection mechanism.

otherwise); O2 optimisation; and Link-Time Optimisation (LTO).

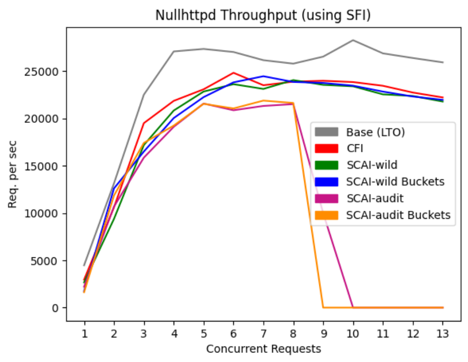
In order to generate accurate results, the webserver received 8 cores to run on through `taskset`. WRK2 was similarly given 32 different cores to limit the interference between the webserver and WRK2 itself. The total utilisation of the RAM memory was never over 2%.

Concretely, we used WRK2 to measure the throughput of Nullhttpd under increasing concurrent requests. For each, we used a Link-Time Optimisation (LTO) compiled version as base level, given that LTO is required for `SCAI` but performs additional optimisation passes. We first compare the LTO version against the webservers as instrumented through CFI to get a baseline for CFI.

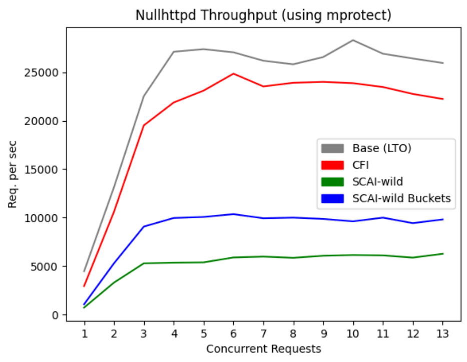
Because `SCAI` requires CFI (as e.g., a ROP would completely nullify the `SCAI` protection), so we would like to know its performance compared to a CFI-only run. In testing `SCAI`, we tested (1) in-the-wild and (2) audit system call lists for protection. Both were tested with and without bucketisation. Note that the `mprotect` protection mechanism has not been tested under the audit system call list, because `mprotect` is not thread-specific as discussed before.

Results. As seen in Figure 6.3, the slowdown of the webservers is marginal when `SCAI` is applied using MPK. CFI-only shows an overhead of 14.8–24.1% compared

[37]See https://github.com/giltene/wrk2, commit 44a94c17.



(a) Throughput when using Software-Fault Isolation.



(b) Throughput when using the `mprotect` system call.

Figure 6.4: Nullhttpd 0.5.0 throughput as a function of concurrency, with non-standard protection mechanisms.

to the LTO baseline. The `SCAI`-wild implementation shows no further overhead without buckets, and an overhead of 3.5% when using buckets. Nullhttpd with `SCAI`-audit saw a larger overhead of 17–24% overhead.

The main bottleneck with Nullhttpd seems to be the dispatcher, which is not instrumented in the wild version, hence the similar overhead to CFI. This shows the power of `SCAI`, since it does not instrument code that does not require it. The audit version does instrument the dispatcher and hence shows a raise in the overhead.

SFI Using SFI, the overhead remains limited as seen in Figure 6.4a. However, after 9–10 concurrent requests, the `SCAI`-audit fails. Here, the secure memory runs out and a subsequent heap memory request fails with an Out-Of-Memory (OOM). This is not considered in the code of nullhttpd and leads to a segmentation fault. This shows a fundamental limitation of SFI, since the SFI mask requires knowledge of the full size and location of the secure memory region a priori.[38]Nonetheless, `SCAI`-wild shows an overhead of 4.3–6.7% while using SFI, and the successful `SCAI`-audit tests show an overhead of 14.7–17.5%. `SCAI`-audit shows that SFI can outperform MPK when its capabilities are invoked frequently — which is more likely the case the more system calls are being instrumented (such as in audit).

[38]Note that the MPK-based solution does not crash with an OOM, since the default behaviour of mspaces is to allocate from a newly requested chunk of heap memory. If used in a real scenario however, it is better to further develop a separate heap manager as mentioned in Section 6.4.

mprot As seen in Figure 6.4b, the mprotect mechanism is significantly slower. SCAI-wild already performs at a 131–286% overhead as shown in the figure above. SCAI-audit has not been tested since it introduces a race-condition that results in false-positives crashing the application.

6.6 Discussion

While SCAI protects against common payloads, it does not protect against attackers that focus on semantics data such as private keys or personal data. SCAI also does not protect against attacks where the data-flow is valid, such as with command injection attacks[39]. The former can be solved with manual annotations inside the source code with additional development effort. The latter cannot be protected with any DFI solution by design. Protection against both attack vectors is left as future work.

Another use of data-flow protection is protecting CFI data, analogous to Code-pointer integrity [116]. This can be implemented using the same MPK-like mechanism, as long as enough MPK keys are at our disposal. When a shadow stack or safe stack is implemented using our MPK implementation, stack variables can be rewritten into a safe stack instead of rewriting them as heap variables. This is theoretically the largest source of the SCAI overhead, so this technique would likely reduce the overall overhead of CFI+SCAI. In our evaluation, we used the default LLVM CFI implementation instead. Note that the CFI implementation used only affected the overhead baseline, since data-only attacks do not interact with CFI mechanisms.

Intel MPK. SCAI relies on Intel MPK to protect its data. When access to Intel MPK is not possible, the mprotect system call or SFI can be used, albeit with their respective downsides. Notably, mprotect is not a per-thread mechanism, so multithreaded applications using the mprotect mechanism can introduce race-conditions that either break the application under normal usage or the converse. SFI on the other hand cannot protect against functionality in dynamic libraries and requires careful usage of the virtual memory space.

[39]Note that these types of attacks are of a different GEN class.

Next Generation Memory Exploits. While `SCAI` protects against data-only attacks, NGMEs are not limited to data-only attacks. In NGMEs, control-flow can be controlled by attackers in CFB attacks, so long as it stays within the allowed CFI targets: this is not protected by `SCAI`. The main problem arises when the data-flow analysis as performed by `SCAI` is invalidated by CFB techniques. This occurs either by longjump-like returns (that are allowed by most CFI implementations); allowed indirect `ret`urn instructions that were not accounted for; or indirect `call` instructions that were not accounted for. All of these could change the context of the executed function, e.g., by writing from non-secure memory into secure memory with an instrumented `memcpy`.

A partial solution would be to override the security-sensitive system call function wrappers and check if their pointer arguments are indeed from secure memory (i.e., with SFI). This can be combined with additional runtime checks to ensure no data-flow is copied from the non-secure memory to the secure memory, which is non-trivial.

A second problem occurs when a secure memory region remains writable for too long. This could occur with e.g., a string format attack on an instrumented `sprintf`. Here, an attacker could jump back more than one stack frame (as allowed due to `longjump`) and keep the MPK region writable for longer, because it is not paired with a `set_readonly` instruction anymore. In this particular example, the AWP is already in the sensitive data-flow so it would not differ anyway, but we cannot guarantee that similar instances *can* break the data-flow protection. A solution that adds a quick read-only MPK property after any `call` instruction is available in our implementation, but has not been tested. The CFB limitation is consistent with related work [142, 27].

Analysis Limitations. In our proof-of-concept implementation of `SCAI` we perform a UD-chain based analysis with a few extra techniques for (indirect) function calls and known data locations. Although this covers the basics, it becomes inaccurate in the complex data-flows often seen in applications. The dynamic library analysis extends the analysis results, but our current implementation lacks pointer analysis such as aliasing [179]. This can cause an under-approximation of the data-flow leading to a false positive, crashing under normal usage. With state-of-the-art static analysis techniques applied, we believe SCAI could become a practical and deployable solution.

`SCAI` does not track asynchronous inter-thread or inter-process communication. This is an open question in the research community [174] but could result in losing track of data-flow information that leads into a security-sensitive system call. On the other hand, no proof-of-concept exploit is known by us that abuses this loss of data-flow information.

By design, we are unable to spot stack spilling. Stack spilling occurs when we do not have enough registers available in hardware when compiling. This means a register-value is pushed on the stack temporarily so the register can be used for something else. This could be an issue in register-sized variables such as the protection bits in `mmap`. Since `SCAI` is implemented as an LLVM compiler pass, it runs the analysis on the LLVM IR. LLVM IR is a static single assignment (SSA) language [158], meaning the amount of registers available is infinite by design. Hence, stack spilling could happen after our implementation has completed its instrumentation. We have not encountered a sensitive data-flow leak in practice.

Finally, we do not detect indirect data-flow. Indirect data-flow is when data is indirectly transferred, e.g., when a number is copied by looping and incrementing. When using indirect data-flow transfers, unprivileged data can be indirectly copied into privileged data, exposing an attack vector. This problem is limited in scope and unlikely to occur in usage of the protected system calls in real-world programs. We have not encountered any real-world examples that contain indirect data-flows into protected system calls.

Data Protection Limitations `SCAI` provides three different mechanisms to protect the security-sensitive data. The data itself is internally handled by DLMalloc and pre-allocated upon initialisation of the application. Our implementation uses a mutex to enforce thread-safety across the secure memory. With additional implementation effort ptmalloc2 could be altered to ensure more efficient thread-safety and scalability of the memory. By managing different thread arenas and MPK keys (in the case of bucketisation) inside ptmalloc2, performance and memory scalability could be increased, less of limitations per technique. In particular, SFI will have a predefined maximum to keep the bitmasks constant (and thus immutable), and mprotect has a maximum amount of contiguous regions of protection bits.

6.7 Conclusion

All system calls are created equal. With respect to security however, some are more equal than others. Concretely, out of 382 system calls available on the latest version of the Linux kernel, a mere *32* of them can be leveraged to gain more control over the underlying system. In practice, only a handful of system calls are used for exploitation. Albeit intuitive, to the best of our knowledge we are the first to systematically analyse and exploit this observation without interfering with normal application behaviour. Our analysis motivated the design of System Call Argument Integrity, the concept of preventing data tampering only for data that is input to a security-sensitive system call.

System Call Argument Integrity creates a secure memory region that is not writable unless strictly necessary. Hence, data-only attacks are naturally blocked unless the vulnerability primitive is already in the target trace. In addition, bucketisation mitigates most of the risk of a sensitive data-flow becoming compromised by creating different memory regions for different system calls. Bucketisation protects all the other system calls from an attacker, even in the event of a system call argument becoming available to an attacker.

Our implementation of System Call Argument Integrity is effective against data-only attacks and `SCAI` is shown to have minimal performance impact (**RQ4**). With either Intel MPK support, single-threaded applications or static linking, `SCAI` works out-of-the-box: no changes are required to the kernel, dynamic libraries or the source code. With additional effort spend on the program analysis aspect, SCAI may become precise enough to be generally deployable on executables.

Case Study	Published	Explicit Primitive	Escalation Primitive
The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86) [166]	2007	✓	`execve`
AEG: Automatic Exploit Generation [8]	2011	✓	`execve`, *shellcodes*
Unleashing Mayhem on Binary Code [29]	2012	✓	*shellcodes*
Control Flow Bending: On the Effectiveness of Control-Flow Integrity [25]	2015	✓	`execve`
Automatic Generation of Data-Oriented Exploits [96]	2015	✓	`execve`, `setuid`
Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity[79]	2015	✓	`execve`
The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later [195]	2017	✓	`execve`, `mprotect`
Modular Synthesis of Heap Exploits [155]	2017	✓	*shellcodes*
Position-independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure [86]	2018	✓	`dup2`, `execve`, `mprotect`, *shellcodes*
Revery: From Proof-of-Concept to Exploitable [202]	2018	×	UNDISCLOSED
Block Oriented Programming: Automating Data-Only Attacks[109]	2018	✓	`execve`
Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters[93]	2019	✓	`execve`

Table 6.1: Privilege Escalation Usage in various Case Studies

7 General Conclusion and Discussion

In the case of a memory violation, even the smallest mistake can have major consequences. Exploiting these memory violations however is a complex, time-consuming process and usually consists of many individual, complicated tasks. Writing an exploit can easily take weeks or months to develop and requires a lot of expertise. The field is ever changing and the next change is due soon with the potential widespread deployment of CFI [119, 120]. Next generation attack techniques show this is not the end [25, 98] even though some protections [142, 27] already propose partial solutions. These attacks *will* be found in the real world after CFI is more widely deployed, as long as the pay-off is large enough. Yet, NGMEs are a hard problem for both attackers and defenders.

Within the research community, we are ahead of the curve. Arguably, we are preparing for the point when the attacks appear "in the wild". Until then, the interest of both the research community and society as a whole will remain limited.

Nonetheless, it is paramount that we continue to stay ahead of the curve. We need to fully understand what attack vectors are still available in NGMEs; find potential dangerous patterns such as two sequential calls to `printf` [25]; understand what aspects can be realistically automated; and find new, better ways of protecting against NGMEs. Only by performing both offensive *and* defensive research can the world be properly prepared against this new, emerging attack vector.

In this dissertation, we started with the overall question:

> **RQ1**: How does the exploitation process work, and how does this apply to NGMEs?

To answer this question, we looked at NGMEs from an offensive perspective. The theory and practice in this dissertation applies mostly to all memory exploits. In other words, memory exploit writing in itself is central. Every aspect is highlighted with respect to its application towards NGMEs, without a loss of generality when applicable to other memory exploits.

Vulnerability **RQ2: What tangible and unambiguous properties does a vulnerability have?** The exploit writing process is broken down in three phases (Chapter 3). At least one vulnerability is required to write an exploit. Vulnerabilities are interesting phenomena that are still vaguely defined and poorly understood to this day. Chapter 4 discusses this and presents an exact definition of a bug, based on measurable properties. To recapitulate, a bug represents a discrepancy between the intended behaviour and the behaviour in practice, or any abstraction layer in between (Section 4.2.1). Any discrepancy can only be attributed to either incorrectness or undefinedness. If the behaviour was properly defined and implemented correctly, there is no space for mistakes. Furthermore, if no discrepancy exists between any two abstraction layers, the behaviour of the application in practice is equal to the intended behaviour and thus no bug exists.

These two properties form the basis of the `GEN` taxonomy. Furthermore, a bug is labelled as a vulnerability if it bears no certainty for the unexploitability of the bug (Section 4.2.3). This definition inevitably remains vague. Ideally, a vulnerability is exploitable with certainty, but assessing exploitability is hard. Although metrics (e.g., CVSS [126]) can determine a certain likelihood of exploitability, the only means of determining exploitability with certainty involves writing the exploit. Not only is this hard and time-consuming (as this dissertation reflects), the process is undecidable: failing to write an exploit provides no guarantee to the absence of any exploit. Not labelling vulnerabilities due to a lack of a working exploit could pose a serious threat as its risk could be underestimated. As such, our definition for a vulnerability turns this around: only a degree of certainty of *un*exploitability removes the vulnerability label from a bug.

From a broader perspective, `GEN` teaches us the true nature of vulnerabilities. It shows where mistakes can be made (the abstraction layers) and explains that mistakes can *only* be a consequence of incorrectness or undefinedness. Instead of definitions that rely on e.g., what is or is not confidential; an asset; or interesting for threat sources, we provide measurable properties in line with the intuitive understanding of vulnerabilities. Less of exploitability, `GEN` removes all debate on whether something is a vulnerability while providing an understanding on *why* it is a vulnerability.

Control **RQ3: Can we find a generic solution to solving HLM problems that is explainable and heap manager agnostic?**
A vulnerability leads to unintended behaviour. Triggering the vulnerability opens up the possibility for the exploit writer to execute more unintended behaviour. In other words, the exploit writer has gained some initial level of control over the application (Section 3.3.1). The subsequent task of the exploit writer is to gain elevated levels of control such as an arbitrary write primitive, Turing-Completeness or even Arbitrary Code Execution. Various techniques and potentially a multitude of vulnerabilities can be used, and various protection schemes (e.g., ASLR or stack canaries) need to be broken or circumvented.

One technique in the case of a heap memory vulnerability is Heap Layout Manipulation. Heap Layout Manipulation provides the attacker to overwrite a value of choice on the heap. Usually this is a data pointer (creating e.g., an AWP) or a control pointer (e.g., to perform CFB). We discussed how to off-load the exploit writer in the HLM task without losing the intuitive understanding of the resulting solution. This led to Hack the Heap, an online puzzle game that accurately represents both synthetic and real-world HLM problems.

Hack the Heap players only require a modern browser and time. No prior knowledge is required, as a series of small tutorials teaches the player all they need. For realistic puzzles, Hack the Heap switches to a backend service that executes the exact operations of the puzzle using a heap manager of choice. Currently, 4 different heap managers are available on https://hacktheheap.io/ but this can be easily extended to different heap managers or OSes.

To play real-world puzzles, we need to generate the puzzles based on real-world behaviour of the vulnerable application. We developed the Hack the Heap recorder to record all heap manager interactions, complete with an interactive console to interact with the application, recognise various operations and more. The recording can be directly turned into a playable puzzle with the HTH refinery.

The full HTH pipeline makes it easier to solve the HLM problem by simplifying and visualising the problem at hand. Alternatively, it can be added to the HTH database so other players can solve the puzzle. Crowd-funding these tasks can provide the exploit writer with multiple solutions where they can choose the most appropriate solution for the remainder of the exploit. If no solution is found, players can mark the puzzle as impossible.

Hopefully, future research continues to consider crowd-sourcing research and gamification. It has been proven useful in other work and we barely touched the surface of its opportunities. We hope that our own work on HLM will be extended, but we also hope that the research direction as a whole takes inspiration from the non-conventional approach.

Payload **RQ4: Can we protect applications against data-only attacks by blocking off common payload targets with a limited performance overhead?**

With the right level of control, exploit writers can achieve their exploitation goals. Goals can range from gathering data (e.g., a private key) to deleting data (e.g., purging a database), and from gaining shell access on the machine to installing a malicious application (e.g., a backdoor). A concrete way of reaching said goal is called a payload (Section 3.4).

Some payload goals (e.g., stealing a private key) are semantically tied to the application under attack. Other payload goals (e.g., gaining a shell) are generic regardless of the semantics of the application[40]. We analysed the generic payloads in the form of shell codes, previously published work and a manual audit on all available Linux system calls (Section 6.2.1). This provided us with a lower and upper bound on all system calls that could be used in exploits. While known to exploit writers intuitively, this analysis specifies clearly what is and what is not a threat. Knowing generic payload targets, we can provide additional protection to ensure these targets are not within reach of exploit writers.

We do so through System Call Argument Integrity (Chapter 6), a technique that utilises Intel MPK functionality to enforce read-only (non-write) access on secure memory regions. These regions contain any data that could flow into system calls as used in common payloads. Any legitimate write is wrapped with two function calls that set the memory region writable — and back to read-only. Across different security-sensitive system calls, different memory regions can be used through

[40]Less of the availability of such functionality in NGMEs.

bucketisation (Section 6.3.3). If one data-flow is already compromised, bucketisation still prevents the attacker from influencing other system call arguments.

System Call Argument Integrity protects against data-only attacks, but could theoretically be circumvented with Control-Flow Bending. We hope that this dissertation shows that NGMEs are hard problems, and protecting against both CFB and Data-only attacks is challenging but necessary. We see System Call Argument Integrity as a stepping stone towards effective, directed protection at a low overhead.

Besides, we show usability of Intel MPK. This could spike both future research into the extended usage of MPK in a security setting. On top, this could convince Intel that MPK is a worthwhile effort to continue, providing a broader range of CPUs with MPK support in the future. In the future, MPK may become a default option on machines to protect against NGMEs.

Future Work

Within the individual contribution chapters, we have discussed various future work angles and directions specific to that body of work. In the overarching notion of this dissertation however, a general direction becomes apparent to evolve the field.

Within the exploitation process itself, the systematisation of chapter 3 could be broadened into a SoK. This could be framed in the concept of weird machines (Section 2.1.1). Here, step 1 would be to find the vulnerability, i.e., entrance to the weird machine. The `GEN` class here depicts the type of weird machine we are considering (the abstraction) in combination with the cause of the existence of such a weird machine. Step 1 also considers choosing what weird machine to enter (where multiple paths could exist that trigger the same vulnerablity with a different environment). Afterwards, programming the weird machine is step 3 (payload), whilst various techniques can be used to make it *easier* to program the weird machines — which is step 2 (control). Control and payload differences are mostly abstracted away with a state machine representation, since control techniques are also part of programming the weird machine within the abstraction. Besides, the payload step narrows down to an end goal, which is a hard problem to formulate within the weird machine representation. These are all challenges that can lead to an SoK contribution with respect to exploit development in weird machine models.

Hack the Heap is more concrete as a research project in itself. In part due to a global pandemic, Hack the Heap has not been evaluated with respect to its enjoyment; and its awareness and educational value. Almost any infosec lecturer I personally spoke to is willing to use Hack the Heap in their course on offensive security, whilst admitting that heap exploitation has thus far been considered too advanced to fit the course. Although this makes it difficult to introduce a control-group to test for effectiveness, We believe Hack the Heap can alter this [37] by simplifying the concept of heap vulnerabilities and their effects. After (part of) the original Hack the Heap tutorial, students can be presented with an actual application that contains a heap vulnerability in combination with the HtH game and the source code. Through solving the puzzle, the aim would be to learn exploit development, heap memory behaviour and memory safety awareness in a broader scale. This can then be evaluated with a questionnaire [143] on the three axes of enjoyment; educational value; and awareness. Awareness is tied to the gamification aspect of HtH and may need additional effort [90]). The questionnaire would be interesting to see paired with the final scores of the students on heap-related questions in the exam, that will actually reflect the intrinsic hard-skills at the end of the course. It it however important that both the game and the questionnaire remain optional for the students not to be motivated beyond what they learn throughout the course material.[41] In this case, Hack the Heap would benefit from implementations of temporal vulnerabilities.

Finally, System Call Argument Integrity needs additional static program analysis techniques and a potential reference monitor to make the false positive rate reasonable. Afterwards, SCAI can be used to implement fine-grained forwards CFI to be used in combination with Intel CETS, just like FineIBT [120].

Conclusion

When CFI gets deployed on a broad scale, it is not to prevent exploitation. It is to deter attackers by making it harder (and thus more costly) to develop an exploit for a given vulnerability, exactly like we have seen with any previous control-level protection scheme (e.g., NX, ASLR or stack canaries). As long as the target is

[41] This is of course also something that should be asked within the questionnaire.

valuable enough in the face of monetisation, information or activism, an exploit *will* be developed. If we add additional protection like CFI, we need to acknowledge that the rules of the game change and prepare appropriately.

The research performed in this dissertation is part of a larger effort into understanding the attack surface before we encounter real-world NGMEs. We broadened the overall understanding of vulnerabilities, discussed how control can be achieved easier through HLM and gamification, and we looked at how to block a data-only attack even if a level of control was successful. We put this together in one framework for exploit development, to understand how exploit development works and find the right approach to deter such exploits.

On a broader spectrum, this dissertation shows that a tremendous amount of work is still to be done in the research community. It is still not known how feasible and broadly applicable NGMEs are, nor is it known how to reliably protect against them. These are both still open questions that need to be solved in the research community before they become a reality.

Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM conference on Computer and communications security*. CCS, 2005.

[2] A. Adams and M. A. Sasse. Users are not the enemy. In *Communications of the ACM*. CACM, 1999.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *IEEE Symposium on Security and Privacy*. S&P, 2008.

[4] aleph1 (Elias Levy). Smashing the stack for fun and profit.

[5] E. Andersen, E. O'Rourke, Y.-E. Liu, R. Snider, J. Lowdermilk, D. Truong, S. Cooper, and Z. Popovic. The impact of tutorials on games of varying complexity. In *SIGCHI Conference on Human Factors in Computing Systems*, 2012.

[6] J. P. Anderson. Computer security technology planning study. In *DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS HQ ELECTRONIC SYSTEMS DIVISION*. ASFC, 1972.

[7] angr. angr/angrop | github. https://github.com/angr/angrop. Accessed: 2021-09-08.

[8] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *Network and Distributed System Security Symposium*. NDSS, 2011.

[9] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith. The page-fault weird machine: Lessons in instruction-less computation. In *Presented as part of the 7th {USENIX} Workshop on Offensive Technologies*. WOOT, 2013.

[10] S. Barnum. Common attack pattern enumeration and classification (CAPEC) schema description. In *MITRE*, 2008.

[11] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. In *IEEE transactions on software engineering*. TSE, 2014.

[12] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating system enhancements to prevent the misuse of system calls. In *ACM conference on Computer and communications security*. CCS, 2000.

[13] B. Biggio, K. Rieck, D. Ariu, C. Wressnegger, I. Corona, G. Giacinto, and F. Roli. Poisoning behavioral malware clustering. In *Workshop on artificial intelligent and security workshop*. AISec, 2014.

[14] R. Bisbey and D. Hollingworth. Protection analysis: Final report. In *Information Sciences Inst*, 1978.

[15] M. Bishop. A taxonomy of unix system and network vulnerabilities. Technical report, Technical Report CSE-95-10, Department of Computer Science, University of California at Davis, 1995.

[16] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security*. ASIACCS, 2011.

[17] BNP Media. Apple failed to disclose security incident affecting 128 million users in 2015. https://www.securitymagazine.com/articles/95179-apple-failed-to-disclose-security-incident-affecting-128-million-users-in-2015. Accessed: 2021-08-24.

[18] J. A. Bockenek, F. Verbeek, P. Lammich, and B. Ravindran. Formal verification of memory preservation of x86-64 binaries. In *Conference on Computer Safety, Reliability, and Security*. CCRS, 2019.

[19] A. Brahmakshatriya, P. Kedia, D. P. McKee, D. Garg, A. Lal, A. Rastogi, H. Nemati, A. Panda, and P. Bhatu. Confllvm: A compiler for enforcing data confidentiality in low-level code. In *EuroSys Conference*. EuroSys, 2019.

[20] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina. Exploit programming: From buffer overflows to weird machines and theory of computation. In *USENIX; login*. login, 2011.

[21] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*. S&P, 2008.

[22] E. Brynjolfsson, J. J. Horton, A. Ozimek, D. Rock, G. Sharma, and H.-Y. TuYe. Covid-19 and remote work: an early look at us data. Technical report, National Bureau of Economic Research, 2020.

[23] N. Burow, X. Zhang, and M. Payer. Sok: Shining light on shadow stacks. In *IEEE Symposium on Security and Privacy*. S&P, 2019.

[24] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. In *ACM Transactions on Information and System Security*. TISSEC, 2008.

[25] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Conference on Security Symposium*. USENIX Sec, 2015.

[26] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*. USENIX Sec, 2014.

[27] S. A. Carr and M. Payer. Datashield: Configurable data confidentiality and integrity. In *ACM Symposium on Information, Computer and Communications Security*. ASIACCS, 2017.

[28] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *symposium on Operating systems design and implementation*. OSDI, 2006.

[29] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*. S&P, 2012.

[30] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM conference on Computer and communications security*. CCS, 2010.

[31] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy*. S&P, 2018.

[32] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*. USENIX Sec, 2005.

[33] Y. Chen and X. Xing. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2019.

[34] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. Yao. Exploitation techniques and defenses for data-oriented attacks. In *IEEE Cybersecurity Development*. SecDev, 2019.

[35] L. Cheng, K. Tian, and D. Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. In *Computer Security Applications Conference*. ACSAC, 2017.

[36] F. Chiarello and M. G. Castellano. Board games and board game design as learning tools for complex scientific concepts: Some experiences. In *International Journal of Game-Based Learning*. IJGBL, 2016.

[37] F. Chiarello and M. G. Castellano. Board games creation as motivating and learning tool for stem. In *European Conference on Game-Based Learning*. ECGBL, 2017.

[38] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in System Dependability*. HotDep, 2009.

[39] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *International Conference*

on Architectural Support for Programming Languages and Operating Systems. ASPLOS, 2011.

[40] V. A. Chris Lattner. The llvm compiler infrastructure. `https://llvm.org/`. Accessed: 2019-09-03.

[41] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *ACM transactions on Programming Languages and Systems*. TOPLAS, 1994.

[42] S. Conrad, J. Clarke-Midura, and E. Klopfer. A framework for structuring learning assessment in a massively multiplayer online educational game: Experiment centered design. In *International Journal of Game-Based Learning*. IJGBL, 2014.

[43] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2015.

[44] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popović, et al. Predicting protein structures with a multiplayer online game. In *Nature*. Nature Publishing Group, 2010.

[45] S. Cooper, A. Treuille, J. Barbero, A. Leaver-Fay, K. Tuite, F. Khatib, A. C. Snyder, M. Beenen, D. Salesin, D. Baker, et al. The challenge of designing scientific discovery games. In *International Conference on the Foundations of Digital Games*, 2010.

[46] J. Corbet. Memory protection keys. https://lwn.net/Articles/643797/. Accessed: 2022-04-11.

[47] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*. USENIX Sec, 1998.

[48] C. Criddle. 'self-driving' cars to be allowed on uk roads this year. https://www.bbc.co.uk/news/technology-56906145. Accessed: 2022-03-14.

[49] CVE-2002-1496. `nullhttpd` 0.5.0 heap overflow `https://www.cvedetails.com/cve/ CVE-2002-1496/`.

[50] CVE-2013-2028. nginx 1.3.9 through 1.4.0 stack overflow `https://www.cvedetails.com/cve/ CVE-2013-2028/`.

[51] CVE-2014-0160. Heartbleed `https://www.cvedetails.com/cve/ CVE-2014-0160/`.

[52] CVE-2014-6271. Shellshock `https://www.cvedetails.com/cve/ CVE-2014-6271/`.

[53] CVE-2019-11839. NginX NJS heap overflow `https://www.cvedetails.com/cve/ CVE-2019-11839/`.

[54] CVE-2019-12206. NginX NJS heap overflow `https://www.cvedetails.com/cve/ CVE-2019-12206/`.

[55] CVE-2019-13617. NginX NJS heap overflow `https://www.cvedetails.com/cve/ CVE-2019-13617/`.

[56] CVE-2021-3962. ImageMagick UaF `https://www.cvedetails.com/cve/ CVE-2021-3962/`.

[57] CVE-2021-44228. Log4j arbitrary code execution `https://www.cvedetails.com/cve/ CVE-2021-44228/`.

[58] J. Dahse, N. Krein, and T. Holz. Code reuse attacks in php: Automated pop chain generation. In *2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2014.

[59] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security*. ASIACCS, 2015.

[60] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*. USENIX Sec, 2014.

[61] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. Cloaker: Hardware supported rootkit concealment. In *IEEE Symposium on Security and Privacy*. S&P, 2008.

[62] T. de Raadt. Exploit Mitigation Tecchniques (in OpenBSD, of course). Presentation, The OpenBSD Project, 2005.

[63] R. Derbyshire, B. Green, D. Prince, A. Mauthe, and D. Hutchison. An analysis of cyber security attack taxonomies. In *IEEE European Symposium on Security and Privacy Workshops*. EuroS&PW, 2018.

[64] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *international conference on Architectural support for programming languages and operating systems*. ASPLOS, 2008.

[65] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*. LCTES, 2003.

[66] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Annual Computer Security Applications Conference*. ACSAC, 2010.

[67] C. Domas. Breaking the x86 isa. In *Black Hat*, 2017.

[68] M. Dominiak and W. Rauner. Efficient approach to fuzzing interpreters. In *BlackHat Asia*, 2019.

[69] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *2015 IEEE Security and Privacy LangSec Workshop*. LANGSEC, 2015.

[70] T. F. Dullien. Weird machines, exploitability, and provable unexploitability. In *IEEE Transactions on Emerging Topics in Computing*. TETC, 2017.

[71] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Heaphopper: Bringing bounded model checking to heap implementation security. In *USENIX Security Symposium*. USENIX Sec, 2018.

[72] N. L. Eisner, O. Barragán, S. Aigrain, et al. Planet Hunters TESS I: TOI 813, a subgiant hosting a transiting Saturn-sized planet on an 84-day orbit. In *Monthly Notices of the Royal Astronomical Society*, 2020.

[73] L. Eliot. Tesla lawsuit over autopilot-engaged pedestrian death could disrupt automated driving progress. https://www.forbes.com/sites/lanceeliot/2020/05/16/lawsuit-against-tesla-for-autopilot-engaged-pedestrian-death-could-disrupt-full-self-driving-progress/. Accessed: 2021-08-24.

[74] P. Engebretson. *The basics of hacking and penetration testing: ethical hacking and penetration testing made easy*. Elsevier, 2013.

[75] S. Engle, S. Whalen, D. Howard, and M. Bishop. Tree approach to vulnerability classification. In *Department of Computer Science, University of California*, 2005.

[76] D. Epple, L. Argote, and R. Devadas. Organizational learning curves: A method for investigating intra-plant transfer of knowledge acquired through learning by doing. In *Organization science*, 1991.

[77] S. Esposito, D. Sgandurra, and G. Bella. Alexa versus alexa: Controlling smart speakers by self-issuing voice commands. In *arXiv preprint arXiv:2202.08619*, 2022.

[78] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point (er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy*. S&P, 2015.

[79] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2015.

[80] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi. Imix: In-process memory isolation extension. In *USENIX Security Symposium*. USENIX Sec, 2018.

[81] D. Fraze. Cyber grand challenge (cgc) (archived). https://www.darpa.mil/program/cyber-grand-challenge. Accessed: 2021-09-28.

[82] J. Gennissen and D. O'Keeffe. Hack the heap: Heap layout manipulation made easy. In *IEEE Security and Privacy Workshops (SPW)*. WOOT, 2022.

[83] M. Gilski, M. Kazmierczyk, S. Krzywda, H. Zábranská, S. Cooper, Z. Popović, F. Khatib, F. DiMaio, J. Thompson, D. Baker, et al. High-resolution structure of a retroviral protease folded as a monomer. In *Acta Crystallographica Section D: Biological Crystallography*. International Union of Crystallography, 2011.

[84] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*. S&P, 2014.

[85] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. Undermining information hiding (and what to do about it). In *USENIX Security Symposium*. USENIX Sec, 2016.

[86] E. Göktas, B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos, and C. Giuffrida. Position-independent code reuse: On the effectiveness of aslr in the absence of information disclosure. In *IEEE European Symposium on Security and Privacy*. EUROS&P, 2018.

[87] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE International Symposium on Security and Privacy*. S&P, 2018.

[88] A. Haggman. Imagining and anticipating cyber futures with games. In *Cyber Threats and NATO 2030: Horizon Scanning and Analysis*, 2020.

[89] A. Haggman et al. Wargaming in cyber security education and awareness training. In *International Journal of Information Security and Cybercrime*. IJISC, 2019.

[90] J. Hamari, J. Koivisto, and H. Sarsa. Does gamification work?–a literature review of empirical studies on gamification. In *international conference on system sciences*. ICSS, 2014.

[91] J. Hamari and J. Tuunanen. Player types: a meta-synthesis. In *Transactions of the Digital Games Research Association*, 2014.

[92] S. Heelan, T. Melham, and D. Kroening. Automatic heap layout manipulation for exploitation. In *USENIX Security Symposium*. USENIX, 2018.

[93] S. Heelan, T. Melham, and D. Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2019.

[94] C. A. R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM*. ACM New York, NY, USA, 1969.

[95] J. D. Howard. An analysis of security incidents on the internet. In *PhD Thesis*. Department of the Air Force Air University, 1997.

[96] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang. Automatic generation of data-oriented exploits. In *USENIX Security Symposium*. USENIX Sec, 2015.

[97] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee. Enforcing unique code target property for control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2018.

[98] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy*. S&P, 2016.

[99] E. M. Hutchins, M. J. Cloppert, R. M. Amin, et al. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. In *Leading Issues in Information Warfare & Security Research*, 2011.

[100] I. E. T. F. (IETF). Rfc4949 (internet security glossary, version 2). `https://tools.ietf.org/html/rfc4949`. Accessed: 2021-09-02.

[101] I. E. T. F. (IETF). Rfc6520 (tls heartbeat extension). `https://tools.ietf.org/html/rfc6520#section-3`. Accessed: 2020-07-28.

[102] V. M. Igure and R. D. Williams. Taxonomies of attacks and vulnerabilities in computer systems. In *IEEE Communications Surveys & Tutorials*. CST, 2008.

[103] S. Ikeda. Record gdpr fine handed to amazon over targeted advertising. https://www.cpomagazine.com/data-protection/record-gdpr-fine-handed-to-amazon-over-targeted-advertising/. Accessed: 2021-08-24.

[104] Intel. Intel® 64 and ia-32 architectures software developer's manual. http://www.intel.com/sdm vol. 3A, 4-36. Accessed: 2022-04-11.

[105] G. Irazoqui, T. Eisenbarth, and B. Sunar. Mascat: Stopping microarchitectural attacks before execution. In *International Association for Cryptologic Research Cryptology Preprint*. IACR, 2016.

[106] ISO. Information technology — Security techniques — Information security management systems — Overview and vocabulary. Standard, International Organization for Standardization, Geneva, CH, Feb. 2018.

[107] ISO. Information technology — Security techniques — Information security risk management. Standard, International Organization for Standardization, Geneva, CH, July 2018.

[108] ISO/IEC. Information technology — Programming languages — C. Standard, International Organization for Standardization, Geneva, CH, July 2018.

[109] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block oriented programming: Automating data-only attacks. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2018.

[110] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *IEEE Symposium on Security and Privacy*. S&P, 2018.

[111] A. Jain. Cve-2021-3156 blogpost: Heap-based buffer overflow in sudo (baron samedit). https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit. Accessed: 2021-08-24.

[112] S. K. Jennifer Hiller. U.s. fuel crisis eases as pipeline returns to normal after hack. https://www.reuters.com/business/energy/massive-replenishment-begins-ease-us-fuel-shortages-2021-05-15/. Accessed: 2021-08-23.

[113] KernelCare. Heartbleed still found in the wild: Did you know that you may be vulnerable? https://linuxize.com/post/heartbleed-still-found-in-the-wild/. Accessed: 2022-03-15.

[114] F. Khatib, S. Cooper, M. D. Tyka, K. Xu, I. Makedon, Z. Popović, and D. Baker. Algorithm discovery by protein folding game players. In *National Academy of Sciences*. National Acad Sciences, 2011.

[115] F. Khatib, F. DiMaio, S. Cooper, M. Kazmierczyk, M. Gilski, S. Krzywda, H. Zabranska, I. Pichova, J. Thompson, Z. Popović, et al. Crystal structure of a monomeric retroviral protease solved by protein folding game players. In *Nature structural & molecular biology*. Nature Publishing Group, 2011.

[116] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation*. OSDI, 2014.

[117] K. Land, A. Slosar, C. Lintott, D. Andreescu, S. Bamford, P. Murray, R. Nichol, M. J. Raddick, K. Schawinski, A. Szalay, et al. Galaxy zoo: the large-scale spin statistics of spiral galaxies in the sloan digital sky survey. In *Monthly Notices of the Royal Astronomical Society*. Blackwell Publishing Ltd Oxford, UK, 2008.

[118] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi. A taxonomy of computer program security flaws. In *ACM Computing Surveys*. CSUR, 1994.

[119] M. Larabel. Clang cfi support upstreamed for linux 5.13 - but only on arm64 for now. https://www.phoronix.com/scan.php?page=news_item&px=Clang-CFI-Linux-5.13. Accessed: 2021-09-08.

[120] M. Larabel. Intel working to combine the best of cet + cfi into "fineibt". https://www.phoronix.com/scan.php?page=news_item&px=Intel-FineIBT-Security. Accessed: 2021-09-08.

[121] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*. USENIX, 2001.

[122] D. Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`. Accessed: 2018-01-11.

[123] K.-s. Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *USENIX Security Symposium*. USENIX, 2002.

[124] X. Li, X. Chang, J. A. Board, and K. S. Trivedi. A novel approach for software vulnerability classification. In *IEEE Reliability and Maintainability Symposium*. RAMS, 2017.

[125] Liming Chen and A. Avizienis. N-version programming: A fault-tolerance approach to rellablllty of software operatlon. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'*. FTCS, 1995.

[126] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. In *IEEE Security & Privacy*. IEEE, 2006.

[127] P. Meunier. Classes of vulnerabilities and attacks. In *Wiley Handbook of Science and Technology for Homeland Security*. Wiley Online Library, 2008.

[128] D. Milmo. Russia unleashed data-wiper malware on ukraine, say cyber experts. https://www.theguardian.com/world/2022/feb/24/russia-unleashed-data-wiper-virus-on-ukraine-say-cyber-experts. Accessed: 2022-03-14.

[129] MITRE. Common weakness enumeration. https://cwe.mitre.org. Accessed: 2020-07-28.

[130] MITRE. Cve terminology. `http://cve.mitre.org/about/terminology.html`. Accessed: 2021-09-02.

[131] MITRE. Cwe glossary. `https://cwe.mitre.org/documents/glossary/index.html#Vulnerability`. Accessed: 2021-09-02.

[132] MITRE. Vulnerability distribution of cve security vulnerabilities by type `https://www.cvedetails.com/vulnerabilities-by-types.php`. Accessed: 2021-12-22.

[133] MITRE. Common weakness scoring system. `http://cwe.mitre.org/cwss/`, 2010. Accessed: 2020-07-28.

[134] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *IEEE Symposium on Security and Privacy*. S&P, 2020.

[135] Musl. musl libc. `https://www.musl-libc.org/`. Accessed: 2019-04-16.

[136] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI, 2009.

[137] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: compiler enforced temporal safety for c. In *International Symposium on Memory management*. ISMM, 2010.

[138] Nergal. The advanced return-into-lib(c) exploits. phrack 11, 58(dec 2001), `http://phrack.com/issues .html?issue=67&id=8`. Accessed: 2022-04-12.

[139] M. Neugschwandtner, A. Sorniotti, and A. Kurmus. Memory categorization: Separating attacker-controlled data. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA, 2019.

[140] NIST. Minimum security requirements for federal information and information systems. Standard, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930, Mar. 2006.

[141] G. Novark and E. D. Berger. Dieharder: securing the heap. In *17th ACM conference on Computer and communications security*. CCS, 2010.

[142] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A.-R. Sadeghi. Hardscope: Hardening embedded systems against data-oriented attacks. In *ACM/IEEE Design Automation Conference*. DAC, 2019.

[143] B. J. Oates. *Researching information systems and computing*. Sage, 2005.

[144] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. Poking holes in information hiding. In *USENIX Security Symposium*. USENIX sec, 2016.

[145] opencpi.org. Open component portability infrastructure. `https://www.opencpi.org/`. Accessed: 2020-07-30.

[146] A. Orendorff. 10 ecommerce trends for 2021: Growth strategies, data & 17 experts on the future of direct-to-consumer (dtc) retail. https://commonthreadco.com/blogs/coachs-corner/ecommerce-trends-future. Accessed: 2021-08-24.

[147] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami. Practical black-box attacks against machine learning. In *ACM Asia conference on computer and communications security*. ASIACCS, 2017.

[148] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *IEEE European symposium on security and privacy*. EuroS&P, 2016.

[149] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA, 2015.

[150] J. Paykin, E. Mertens, M. Tullsen, L. Maurer, B. Razet, A. Bakst, and S. Moore. Weird machines as insecure compilation. In *IEEE Workshop on Foundations of Computer Security*. FCS, 2019.

[151] J. Pewny, P. Koppe, and T. Holz. Steroids for doped applications: A compiler for automated data-oriented programming. In *IEEE European Symposium on Security and Privacy*. EuroS&P, 2019.

[152] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *IEEE Symposium on Security & Privacy*. S&P, 2020.

[153] E. Poll. Langsec revisited: input security flaws of the second kind. In *2018 IEEE Security and Privacy LangSec Workshop*. LANGSEC, 2018.

[154] M. Prensky. Digital game-based learning. *ACM Computers in Entertainment*, 2003.

[155] D. Repel, J. Kinder, and L. Cavallaro. Modular synthesis of heap exploits. In *Workshop on Programming Languages and Analysis for Security*. PLAS, 2017.

[156] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis. Revisiting browser security in the modern era: New data-only attacks and defenses. In *IEEE European Symposium on Security and Privacy*. EUROS&P, 2017.

[157] J. Roney, T. Appel, P. Pinisetti, and J. Mickens. Identifying valuable pointers in heap data. In *IEEE Security and Privacy Workshops (SPW)*. WOOT, 2021.

[158] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL, 1988.

[159] saaramar. Deterministic lfh. https://github.com/saaramar/Deterministic_LFH. Accessed: 2022-02-22.

[160] J. Salwan. Ropgadget - gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/. Accessed: 2021-09-08.

[161] J. Salwan. Shellcodes database for study cases. `http://shell-storm.org/shellcode/`. Accessed: 2019-03-14.

[162] K. Scarfone and P. Mell. An analysis of cvss version 2 vulnerability scoring. In *International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2009.

[163] M. Schink and J. Obermaier. Taking a look into {Execute-Only} memory. In *USENIX Workshop on Offensive Technologies*. WOOT, 2019.

[164] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. In *Black Hat*. UBM, 2015.

[165] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX conference on Security*. USENIX Sec, 2010.

[166] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security*. CCS, 2007.

[167] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy*. S&P, 2016.

[168] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2017.

[169] C. Simmons, C. Ellis, S. Shiva, D. Dasgupta, and Q. Wu. Avoidit: A cyber attack taxonomy. In *Annual Symposium on Information Assurance*. ASIA, 2014.

[170] L. Simon, D. Chisnall, and R. Anderson. What you get is what you c: Controlling side effects in mainstream c compilers. In *IEEE European Symposium on Security and Privacy*. EuroS&P, 2018.

[171] Solar Designer. Getting around non-executable stack (and fix). Aug. 1997.

[172] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. Hdfi: Hardware-assisted data-flow isolation. In *IEEE Symposium on Security and Privacy*. S&P, 2016.

[173] A. Sotirov. Heap feng shui in javascript. In *Black Hat Europe*, 2007.

[174] P. S. Souza, S. S. Souza, M. G. Rocha, R. R. Prado, and R. N. Batista. Data flow testing in concurrent programs with message passing and shared memory paradigms. In *Procedia Computer Science*. Elsevier, 2013.

[175] E. H. Spafford. The internet worm program: An analysis. In *ACM SIGCOMM Computer Communication Review*. ACM, 1989.

[176] B. Spengler. Pax: The guaranteed end of arbitrary code execution. https://grsecurity.net/PaX-presentation_files/frame.htm. Accessed: 2022-02-21.

[177] Statista. Installed base of smart speakers worldwide in 2020 and 2024. https://www.statista.com/statistics/878650/worldwide-smart-speaker-installed-base-by-country/. Accessed: 2021-08-24.

[178] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*. NDSS, 2016.

[179] Y. Sui and J. Xue. Svf: interprocedural static value-flow analysis in llvm. In *ACM International conference on compiler construction*. CC, 2016.

[180] Z. Sun, B. Feng, L. Lu, and S. Jha. Oei: Operation execution integrity for embedded devices. In *arXiv preprint arXiv:1802.03462*. arXiv, 2018.

[181] Pax: The guaranteed end of arbitrary code execution. https://wiki.osdev.org/System_V_ABI. Accessed: 2022-03-05.

[182] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *IEEE Symposium on Security and Privacy*. S&P, 2013.

[183] T. C. Team. Clang documentation - undefined behavior sanitizer. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`. Accessed: 2020-07-30.

[184] The Center for Internet Security. Solaris benchmark v1.3.0. standard, The Center for Internet Security, 2004.

[185] The Clang Team. Safestack. https://clang.llvm.org/docs/SafeStack.html. Accessed: 2022-03-18.

[186] The Register. Intel ships 'execute disable' pentium 4s. https://www.theregister.com/2004/10/04/intel_nx_p4s/. Accessed: 2021-09-06.

[187] J. Thompson. Six facts about address space layout randomization on windows. https://www.fireeye.com/blog/threat-research/2020/03/six-facts-about-address-space-layout-randomization-on-windows.html. Accessed: 2021-09-08.

[188] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*. USENIX Sec, 2014.

[189] torvalds/linux. Linux master branch x86-64 syscall table. `https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64 .tbl`. Accessed: 2018-08-29.

[190] K. Tsipenyuk, B. Chess, and G. McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. In *IEEE Security & Privacy*. S&P, 2005.

[191] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In *London Mathematical Society*, 1937.

[192] A. Turner. August 2021 mobile user statistics: Discover the number of phones in the world & smartphone penetration by country or region. https://www.bankmycell.com/blog/how-many-phones-are-in-the-world. Accessed: 2021-08-24.

[193] C. Valasek. Understanding the low fragmentation heap. In *Blackhat USA*, 2010.

[194] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive cfi. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2015.

[195] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2017.

[196] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM SIGSAC conference on computer and communications security*. CCS, 2016.

[197] J. Vanegue. The weird machines in proof-carrying code. In *2014 IEEE Security and Privacy LangSec Workshop*. LANGSEC, 2014.

[198] F. Verbeek, J. Bockenek, A. Bharadwaj, B. Ravindran, and I. Roessle. Establishing a refinement relation between binaries and abstract code. In *ACM-IEEE International Conference on Formal Methods and Models for System Design*. MEMOCODE, 2019.

[199] L. Von Ahn and L. Dabbish. Labeling images with a computer game. In *SIGCHI conference on Human factors in computing systems*, 2004.

[200] L. Von Ahn, R. Liu, and M. Blum. Peekaboom: a game for locating objects in images. In *SIGCHI conference on Human Factors in computing systems*, 2006.

[201] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*. SOSP, 1993.

[202] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. Revery: From proof-of-concept to exploitable. In *ACM SIGSAC Conference on Computer and Communications Security*. CCS, 2018.

[203] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou. {MAZE}: Towards automated heap feng shui. In *USENIX Security*. USENIX, 2021.

[204] Y. Wei, S. Luo, J. Zhuge, J. Gao, E. Zheng, B. Li, and L. Pan. Arg: Automatic rop chains generation. In *IEEE Access*, 2019.

[205] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Workshop on Memory Management*. WMM, 1995.

[206] I. Yun, D. Kapil, and T. Kim. Automatic techniques to systematically discover new heap exploitation primitives. In *USENIX Security Symposium*. USENIX Sec, 2020.

[207] M. Zalewski. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`. Accessed: 2018-08-02.

[208] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*. S&P, 2013.

[209] T. Zhang, Y. Zhang, and R. B. Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. RAID, 2016.

[210] V. Zimmermann and K. Renaud. Moving from a 'human-as-problem" to a 'human-as-solution" cybersecurity mindset. In *International Journal of Human-Computer Studies*. IJHCS, 2019.

Appendices

A GEN classification quickstart

Classifying a given bug using GEN requires a few details on the situation. In short, we follow the three-step process from Picture 4.1: (1) determine the abstraction layers involved, (2) determine whether it's due to incorrectness or undefinedness, and (3) attempt to determine the harm it can cause. We always consider the erroneous behaviour that arises from the bug when determining its class, irrespective of any other behaviour (that e.g., may exhibit more erroneous behaviour). This uses the classification sheet as seen on the next page.

Abstraction Layers. Following the abstraction layers, we first determine whether optional layers are involved: see the second column. For every crossed out optional layer, we can cross these out and cross out the respective discrepancy options. Then we go through each abstraction layer to check for discrepancies (right-hand side mark).

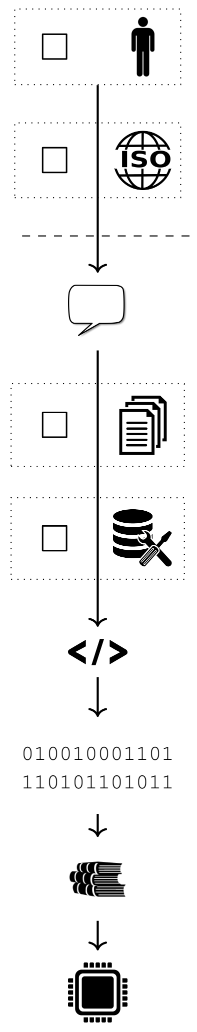
As soon as the behaviour differs across two layers, we tick the box between the two layers and check what potential labels could be involved. Note that going from "undefined behaviour" to "undefined behaviour" does not pose a change in behaviour of the two layers, and hence will not be considered a bug (yet) — this will be exposed through the undefinedness property as discussed in Section 4.3.2. If no two layers can be found with a behavioural discrepancy, then what we are analysing is not a bug, as hardware execution does exactly what the original description told it to do. We do this until all layers have been considered — even when a discrepancy has been found.

Definedness. Given all layers, we question whether the higher layer of the two has defined behaviour in the erroneous example. For this, use the left-hand checkboxes on the classification sheet. If so, it is an incorrectness issue. If it is undefined, there is an undefinedness issue. In combination with the discrepancy location, this provides the bug with a unique class.

Vulnerability. Can we prove beyond doubt that the bug does *not* expose any previously inaccessible data or functionality? If so, then we mark it a bug, preferably while giving the full reasoning on why this cannot be used within an exploit. Alternatively, we mark it a vulnerability, to not provide false information on the nature of a potentially harmful bug. This concludes the classification.

GEN Classification Sheet

Name:	Application:	GEN ID:
CVE ID:	CWE ID:	CVSS Vector:



Defined?	Relevant?	
☑	☐	Human-Computer Interaction (Z)
☑	☐	Security Policy (P)
☐		Informal Description (A)
☐	☐	Formal Specification (B)
☐	☐	Data & Setup (C)
☐		Source Code (D)
☐		Machine Code (E)
☐		Process (F)
☑		Hardware Execution (G)

Discrepancy?		
AA-I	☐	If informal description is inconsistent or contradictory
ZA-I	☐	HCI w.r.t. informal description
PA-I	☐	If informal description violates the security Policy
AB-IU	☐	Only if formal spec. available
AC-IU BC-IU	☐	Only if data / setup applicable
AD-IU BD-IU CD-IU	☐	If source code has a discrepancy with the next available above
DE-IU	☐	
EF-IU	☐	
FG-IU	☐	

☐ Is a vulnerability

Additional comments:

B Table of bugs found with AFL, labelled with GEN and CWE

Product	CVE	Description	Category	CWE Class
libjpeg-turbo	CVE-2013-6630	read of uninitialised memory	Source-Machine-U	CWE-189
libpng	CVE-2015-0973	Heap Overflow	Source-Machine-U	CWE-119
libtiff	CVE-2014-8127	Out-of-bounds read	Source-Machine-U	CWE-125
libtiff	CVE-2014-8128	Out-of-bounds write	Source-Machine-U	CWE-787
libtiff	CVE-2014-8129	Out-of-bounds write	Source-Machine-U	CWE-787
mozjpeg	N/A	Out-of-bounds write	Source-Machine-U	CWE-190*
PHP	CVE-2015-0232	Uninitialised pointer use	Source-Machine-U	CWE-824
PHP	N/A	Null-pointer dereference	Source-Machine-U	CWE-476*
PHP	CVE-2015-0231	Use-after-free	Source-Machine-U	CWE-416
PHP	N/A	Use-after-free	Source-Machine-U	CWE-416*
PHP	N/A	Out-of-bounds read	Source-Machine-U	CWE-125*
PHP	CVE-2017-5340	Uninitialised memory access	Source-Machine-U	CWE-190
Mozilla Firefox	CVE-2014-1564	Uninitialised memory access	Source-Machine-U	CWE-824
Mozilla Firefox	CVE-2014-1580	Uninitialised memory access	Source-Machine-U	CWE-200
Mozilla Firefox	CVE-2014-8637	Uninitialised memory access	Source-Machine-U	CWE-200
Internet Explorer	CVE-2015-0061	Uninitialised memory access	Source-Machine-U	CWE-200
Internet Explorer	CVE-2015-0080	Uninitialised memory access	Source-Machine-U	CWE-200
Internet Explorer	CVE-2015-0076	Uninitialised memory access	Source-Machine-U	CWE-200
PCRE	CVE-2015-0323	Heap Overflow	Source-Machine-U	CWE-119
PCRE	CVE-2015-8380	Heap Overflow	Source-Machine-U	CWE-119
PCRE	N/A	Stack Corruption	Source-Machine-U	CWE-193*
Adobe Reader	CVE-2016-4198	Out-of-bounds write	Source-Machine-U	CWE-119
Adobe Reader	CVE-2016-6969	Use-after-free	Source-Machine-U	CWE-416
Adobe Reader	CVE-2016-6978	Out-of-bounds read	Source-Machine-U	CWE-119

Product	CVE	Description	Category	CWE Class
OpenSSL	CVE-2015-0289	NULL pointer dereference	Source-Machine-U	CWE-476
OpenSSL	CVE-2015-1790	NULL pointer dereference	Source-Machine-U	CWE-476
OpenSSL	CVE-2015-1789	Out-of-bounds read	Source-Machine-U	CWE-119
OpenSSL	CVE-2015-3195	Memory Leak	Source-Machine-U	CWE-200
OpenSSL	N/A	Incorrect Results	(in)Formal-Source-I[1]	CWE-192*
OpenSSL	CVE-2016-2108	Buffer Underflow	Source-Machine-U[2]	CWE-119
libwpd	N/A	Endless Loop	Unknown	Unknown
libpagemaker	N/A	Undisclosed	Unknown	Unknown
poppler	N/A	Out-of-bounds read	Source-Machine-U	CWE-125*
Freetype	N/A	Buffer overflow	Source-Machine-U	CWE-120*
Freetype	N/A	Off-by-one error	Source-Machine-U	CWE-193*
GnuTLS	CVE-2014-8564	Out-of-bounds write	Source-Machine-U	CWE-310
Libksba	CVE-2014-9087	Buffer overflow	Source-Machine-U	CWE-191
GnuPG	N/A	NULL dereference (multiple)	Source-Machine-U	CWE-476*
GnuPG	CVE-2015-1606	Use-after-free	Source-Machine-U	CWE-416
GnuPG	CVE-2015-1607	Invalid read operation	Source-Machine-U	CWE-20
OpenSSH	N/A	NULL pointer dereference	Source-Machine-U	CWE-476*
OpenSSH	N/A	NULL pointer dereference	Source-Machine-U	CWE-476*
OpenSSH	N/A	Undisclosed	Unknown	Unknown
OpenSSH	N/A	Undisclosed	Unknown	Unknown
PuTTY	CVE-2015-5309	Integer Overflow	Source-Machine-U	CWE-189
PuTTY	N/A	Erasing out-of-bounds memory	Source-Machine-U	CWE-404*
ntpd	CVE-2015-7855	Abort instead of returning error	Informal-Source-I	CWE-20
ntpd	N/A	NULL pointer dereference	Source-Machine-U	CWE-476*
nginx	N/A	Self-dependent streams accepted	Informal-Source-U	CWE-1047*
nginx	N/A	Stack overflow	Source-Machine-U	CWE-120*
nginx	N/A	Double free	Source-Library-U	CWE-415*
bash	CVE-2014-6277	Uninitialised memory access	Source-Machine-U	CWE-78
bash	CVE-2014-6278	Improper parsing	Informal-Source-I	CWE-78

Product	CVE	Description	Category	CWE Class
tcpdump	CVE-2014-8768	Integer underflow	Source-Machine-U	CWE-191
tcpdump	CVE-2014-8767	Integer underflow	Source-Machine-U	CWE-189
tcpdump	CVE-2014-8769	Out-of-bounds read	Source-Machine-U	CWE-119
tcpdump	N/A	Incorrect type conversion	Source-Machine-U	CWE-843*
tcpdump	N/A	Out-of-bounds read (×2)	Source-Machine-U	CWE-125*
tcpdump	CVE-2015-3138	Out-of-bounds read	Source-Machine-U	CWE-20
tcpdump	CVE-2016-7993	Buffer overflow	Source-Machine-U	CWE-119
JavaScriptCore	N/A	Multiple(4) Assertion Failures	Informal-Source-IU	CWE-436*
pdfium	N/A	Invalid pointer dereference	Source-Machine-U	CWE-822*
ffmpeg	N/A	Use-after-free	Source-Machine-U	CWE-416*
ffmpeg	N/A	Array out-of-bounds access	Source-Machine-U	CWE-129*
ffmpeg	N/A	Uninitialised memory use	Source-Machine-U	CWE-908*
ffmpeg	N/A	Out-of-bounds read	Source-Machine-U	CWE-125*
libarchive	CVE-2015-8915	Invalid memory read	Source-Machine-U	CWE-125
libarchive	CVE-2015-8916	NULL pointer dereference	Source-Machine-U	CWE-476
libarchive	CVE-2015-8917	Invalid memory read	Source-Machine-U	CWE-476
libarchive	CVE-2015-8918	Overlapping memcpy	Source-Machine-U	CWE-119
libarchive	CVE-2015-8919	Heap out-of-bounds read	Source-Machine-U	CWE-119
libarchive	CVE-2015-8920	Stack out-of-bounds read	Source-Machine-U	CWE-125
libarchive	CVE-2015-8921	Global out-of-bounds read	Source-Machine-U	CWE-125
libarchive	CVE-2015-8922	NULL pointer dereference	Source-Machine-U	CWE-476
libarchive	CVE-2015-8923	Undisclosed	Unknown	CWE-20
libarchive	CVE-2015-8924	Heap out-of-bounds read	Source-Machine-U	CWE-125
libarchive	CVE-2015-8925	Invalid memory read	Source-Machine-U	CWE-125

Product	CVE	Description	Category	CWE Class
libarchive	CVE-2015-8926	NULL pointer dereference	Source-Machine-U	CWE-476
libarchive	CVE-2015-8927	Heap out-of-bounds read	Source-Machine-U	CWE-125
libarchive	CVE-2015-8928	Heap out-of-bounds read	Source-Machine-U	CWE-125
libarchive	CVE-2015-8929	Memory leak	Source-Machine-U	CWE-119
libarchive	CVE-2015-8931	Signed Integer Overflow	Source-Machine-U	CWE-190
libarchive	CVE-2015-8932	Invalid Shift-left	Source-Machine-U	CWE-20
libarchive	CVE-2015-8930	Endless loop	Informal-Source-I	CWE-20
libarchive	CVE-2015-8933	Signed Integer Overflow	Source-Machine-U	CWE-190
libarchive	CVE-2015-8934	Heap out-of-bounds read	Source-Machine-U	CWE-125
Wireshark	N/A	Array overflow	Source-Machine-U	CWE-129*
Wireshark	N/A	Heap overflow	Source-Machine-U	CWE-122*
Wireshark	N/A	Incorrect type conversion	Source-Machine-U	CWE-681*
ImageMagick	N/A	Underflow / Overflow	Source-Machine-U	CWE-191*
ImageMagick	CVE-2016-5687	Out-of-bounds read	Source-Machine-U	CWE-125
ImageMagick	CVE-2016-5688	Undisclosed	Unknown	CWE-119
ImageMagick	CVE-2016-5689	NULL pointer dereference	Source-Machine-U	CWE-476
ImageMagick	CVE-2016-5690	NULL pointer dereference	Source-Machine-U	CWE-476
ImageMagick	CVE-2016-5688	Lack of Validation	Unknown	CWE-119
ISC BIND	CVE-2015-5477	Assertion Failure	Informal-Source-I	CWE-19
ISC BIND	CVE-2015-5722	Assertion Failure	Informal-Source-I	CWE-20
ISC BIND	CVE-2015-5986	Assertion Failure	Informal-Source-I	CWE-20
QEMU	N/A	Out-of-bounds write	Source-Machine-U	CWE-787*
QEMU	N/A	Uninitialised function pointer	Source-Machine-U	CWE-824*
lcms	N/A	Stack overflow	Source-Machine-U	CWE-120*

[1] This was in a test combining AFL with N-version programming. [2] The underflow is a result of two separate vulnerabilities of this type.

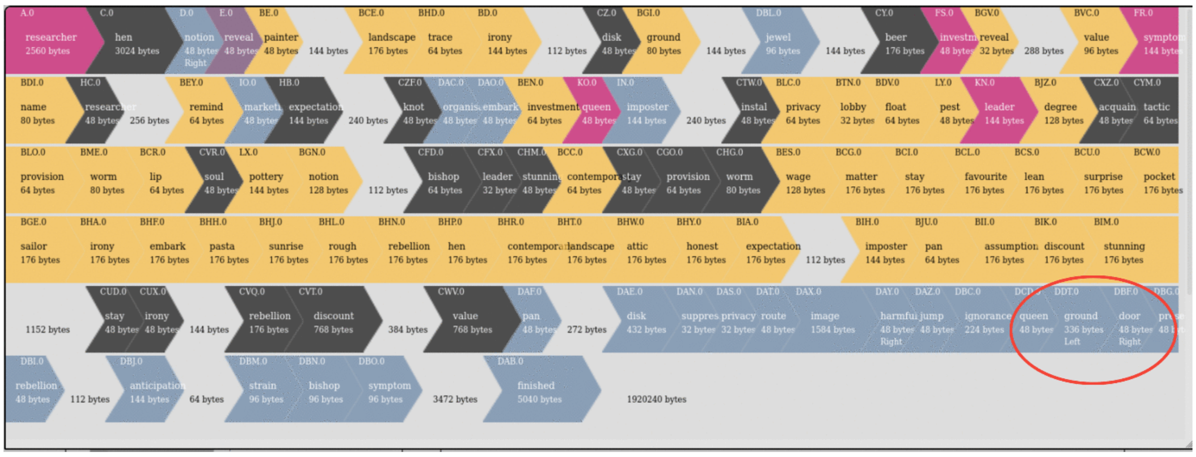
C Context-free Grammar of the puzzle format

This is the context-free grammar used to represent a HTH puzzle in string format. It starts with a magic (`HPM2/`) followed by the heap manager F (e.g., first fit, ptmalloc, jemalloc). Afterwards, it describes the attack type A (e.g., OFA meaning overflow on allocation) and the size of the heap Z. This is followed by the individual operations O, with a name, colon(:) and a list of actions. Actions T are described by a number (e.g., 1 is a `malloc`, etc.) a tag to represent its trace followed by a name and argument(s) if applicable. Different operations are separated with a period(.), an additional period suggests that the operation is an initialisation operation.

```
S -> 'HPM2/'FAZO
F -> 'F'|'B'|'L'|'N'|'P'|'D'|'T'|'J'
A -> 'OVF'|'OFA'|'OFD'|..
Z -> [num]'T'
O -> O.O
  | [name]:T
  | .[name]:T
T -> TT
  | [0-4][TAG]P(R)
P -> &B
  | &T
  | λ
R -> [name]:[ARG]
  | [name]:
  | λ
[ARG] -> [ARG],[ARG]
      | [num]
[TAG] -> [A-Z]+
[name] -> [a-zA-Z]+
[num] -> [0-9]+
```

D Heap Layout solution from CVE-2019-11839

This figure shows the full layout of the heap of CVE-2019-18839 after performing the solution as discussed in Section 5.4.2. At the bottom right, two adjacent puzzle pieces are highlighted with a red ellipse. These "left" and "right" pieces are the bugged and target pieces respectively, depicting a successful solution to the HLM problem.



E Comparison between HTH and an enumeration approach

When provided with a puzzle, it is difficult to compare the quality of the users playing against other techniques. We can however estimate the time it takes for either a depth-first search on the available operations or a random search. Estimating this precisely is not straightforward however due to dependencies: any operation that enables or disables another operation changes the options for the next choice of operation.

In the puzzle for CVE-2019-11839, we have a total of 16 base operations (that can be performed at any time). One operation (creating an array) enables a chain of operations (growing the array), each of which disable an operation. Creating two arrays would also create two additional operations. If we are able to keep growing our array (which is not in the current puzzle), the amount of operations can be described in the following formula.

$$\Sigma_{i=1}^{n}(x+i-1)^{n-i}(x+i)$$

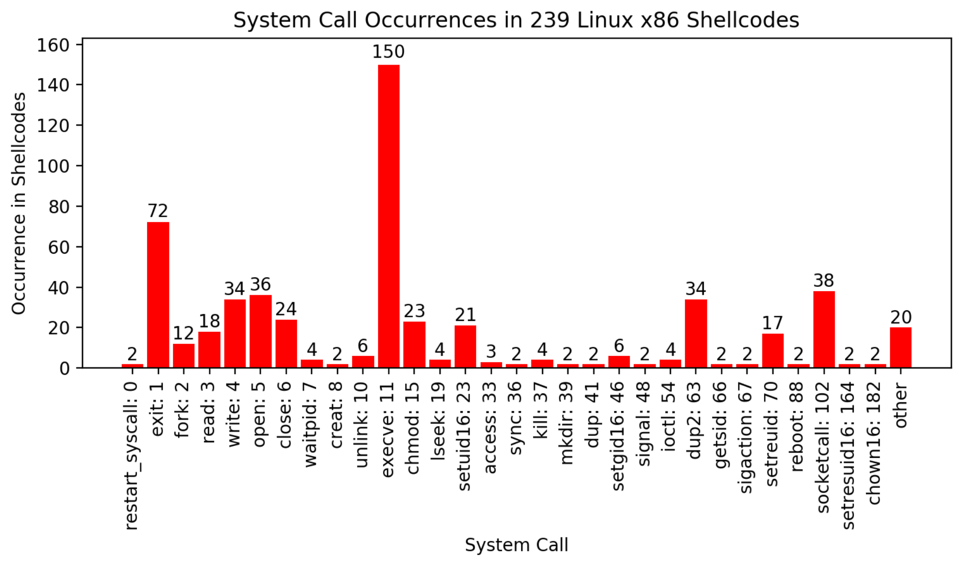
Here, n denotes the amount of operations performed in the sequence, and x denotes the amount of non-changing operations: $x = 15$ in CVE-2019-11839 and $x = 16$ in CVE-2019-12206. If we check for a correct solution after every operation, we do not need to repeat smaller chains. The table below shows the time it takes to enumerate all options of a given length (after compensating for non-infinite growth). Here we assume that we can perform an operation followed by a check if the heap layout has been achieved 8 times per second per CPU, including all heap manager logic and ignoring the time it takes to perform the initial operation, which generally takes a significant amount of time. As the table shows, a full enumeration would take in the worst case approximately 6 CPU-days and 168 CPU-days for CVE-2019-11839 and CVE-2019-12206 respectively, and on average at least 3 and 84 CPU-days respectively.

# of operations performed max	1	2	3	4	5	6	7
CVE-2019-11839	2s	64s	24m	8h	**6d**	117d	2041d
CVE-2019-12206	2s	72s	29m	10h	9d	**168d**	3168d

Approximate CPU core time (in **s**econds; **m**inutes; **h**ours and **d**ays) to enumerate all possibilities for the given puzzles, when 8 full operations and heap layout checks can be performed per second. Highlighted are the set where our solution appear.

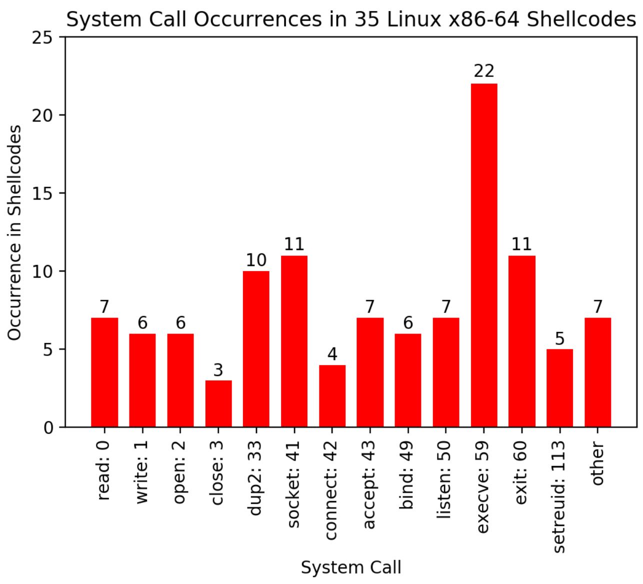
F Shellcode System Call Occurrences

F.1 32-bit Shellcode System Calls



x86 (32-bit) shellcode syscall imbalance. *Other* refers to the following: `chdir`, `getuid`, `stime`, `pause`, `rmdir`, `pipe`, `geteuid16`, `umount`, `setpgid`, `chroot`, `getppid`, `sethostname`, `old_readdir`, `iopl`, `wait4`, `ipc`, `setdomainname`, `nanosleep`, `pwrite64` and `lstat64`, each occurring exactly once.

F.2 64-bit Shellcode System Calls



x86-64 shellcode syscall imbalance. *Other* refers to the following: `access`, `kill`, `setuid`, `setregid`, `reboot`, `sethostname` and `execveat`, each occurring exactly once.

G System Call Audit

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	0	0x0	sys_read	unsigned int fd	char *buf	size_t count			
🟨	1	0x1	sys_write	unsigned int fd	const char *buf	size_t count			
🟥🟨	2	0x2	sys_open	const char *file-name	int flags	int mode			
🟩	3	0x3	sys_close	unsigned int fd					
🟩	4	0x4	sys_stat	const char *file-name	struct stat *statbuf				

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟩	5	0x5	sys_fstat	unsigned int fd	struct stat *statbuf				
🟩	6	0x6	sys_lstat	fconst char *filename	struct stat *statbuf				
🟩	7	0x7	sys_poll	struct poll_fd *ufds	unsigned int nfds	long timeout_msecs			
🟨	8	0x8	sys_lseek	unsigned int fd	off_t offset	unsigned int origin			
🟥	9	0x9	sys_mmap	unsigned long addr	unsigned long len	unsigned long prot	unsigned long flags	unsigned long fd	unsigned long off
🟥	10	0xa	sys_mprotect	unsigned long start	size_t len	unsigned long prot			
🟩	11	0xb	sys_munmap	unsigned long addr	size_t len				
🟩	12	0xc	sys_brk	unsigned long brk					
🟥	13	0xd	sys_rt_sigaction	int sig	const struct sigaction *act	struct sigaction *oact	size_t sigsetsize		
🟩	14	0xe	sys_rt_sigprocmask	int how	sigset_t *nset	sigset_t *oset	size_t sigsetsize		
🟩	15	0xf	sys_rt_sigreturn	unsigned long __unused					
🟥	16	0x10	sys_ioctl	unsigned int fd	unsigned int cmd	unsigned long arg			
🟨	17	0x11	sys_pread64	unsigned long fd	char *buf	size_t count	loff_t pos		
🟨	18	0x12	sys_pwrite64	unsigned int fd	const char *buf	size_t count	loff_t pos		
🟨	19	0x13	sys_readv	unsigned long fd	const struct iovec *vec	unsigned long vlen			

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	20	0x14	sys_writev	unsigned long fd	const struct iovec *vec	unsigned long vlen			
🟩	21	0x15	sys_access	const char *filename	int mode				
🟨	22	0x16	sys_pipe	int *filedes					
🟩	23	0x17	sys_select	int n	fd_set *inp	fd_set *outp	fd_set*exp	struct timeval *tvp	
🟩	24	0x18	sys_sched_yield						
🟩	25	0x19	sys_mremap	unsigned long addr	unsigned long old_len	unsigned long new_len	unsigned long flags	unsigned long new_addr	
🟨	26	0x1a	sys_msync	unsigned long start	size_t len	int flags			
🟩🟧	27	0x1b	sys_mincore	unsigned long start	size_t len	unsigned char *vec			
🟩	28	0x1c	sys_madvise	unsigned long start	size_t len_in	int behavior			
🟩	29	0x1d	sys_shmget	key_t key	size_t size	int shmflg			
🟥	30	0x1e	sys_shmat	int shmid	char *shmaddr	int shmflg			
🟩	31	0x1f	sys_shmctl	int shmid	int cmd	struct shmid_ds *buf			
🟨	32	0x20	sys_dup	unsigned int fildes					
🟨	33	0x21	sys_dup2	unsigned int oldfd	unsigned int newfd				
🟩	34	0x22	sys_pause						
🟩	35	0x23	sys_nanosleep	struct timespec *rqtp	struct timespec *rmtp				
🟩	36	0x24	sys_getitimer	int which	struct itimerval *value				
🟩	37	0x25	sys_alarm	unsigned int seconds					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟩	38	0x26	sys_setitimer	int which	struct itimerval *value	struct itimerval *ovalue			
🟩	39	0x27	sys_getpid						
🟨	40	0x28	sys_sendfile	int out_fd	int in_fd	off_t *offset	size_t count		
🟥🟨	41	0x29	sys_socket	int family	int type	int protocol			
🟨	42	0x2a	sys_connect	int fd	struct sockaddr *uservaddr	int addrlen			
🟨	43	0x2b	sys_accept	int fd	struct sockaddr *upeer_sockaddr	int *upeer_addrlen			
🟨	44	0x2c	sys_sendto	int fd	void *buff	size_t len	unsigned flags	struct sockaddr *addr	int addr_len
🟨	45	0x2d	sys_recvfrom	int fd	void *ubuf	size_t size	unsigned flags	struct sockaddr *addr	int *addr_len
🟨	46	0x2e	sys_sendmsg	int fd	struct msghdr *msg	unsigned flags			
🟨	47	0x2f	sys_recvmsg	int fd	struct msghdr *msg	unsigned int flags			
🟦🟧	48	0x30	sys_shutdown	int fd	int how				
🟨	49	0x31	sys_bind	int fd	struct sockaddr *umyaddr	int addrlen			
🟨	50	0x32	sys_listen	int fd	int backlog				
🟩	51	0x33	sys_getsockname	int fd	struct sockaddr *usockaddr	int *usockaddr_len			
🟩	52	0x34	sys_getpeername	int fd	struct sockaddr *usockaddr	int *usockaddr_len			
🟥🟨	53	0x35	sys_socketpair	int family	int type	int protocol	int *usockvec		

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	54	0x36	sys_setsockopt	int fd	int level	int optname	char *optval	int optlen	
🟩	55	0x37	sys_getsockopt	int fd	int level	int optname	char *optval	int *optlen	
🟥	56	0x38	sys_clone	unsigned long clone_flags	unsigned long newsp	void *parent_tid	void *child_tid		
🟩🟧	57	0x39	sys_fork						
🟩🟧	58	0x3a	sys_vfork						
🟥	59	0x3b	sys_execve	const char *filename	const char *const argv[]	const char *const envp[]			
🟨	60	0x3c	sys_exit	int error_code					
🟩	61	0x3d	sys_wait4	pid_t upid	int *stat_addr	int options	struct rusage *ru		
🟨🟧	62	0x3e	sys_kill	pid_t pid	int sig				
🟩🟧	63	0x3f	sys_uname	struct old_utsname *name					
🟩	64	0x40	sys_semget	key_t key	int nsems	int semflg			
🟩🟧	65	0x41	sys_semop	int semid	struct sembuf *tsops	unsigned nsops			
🟩🟧	66	0x42	sys_semctl	int semid	int semnum	int cmd	union semun arg		
🟩	67	0x43	sys_shmdt	char *shmaddr					
🟩	68	0x44	sys_msgget	key_t key	int msgflg				
🟩	69	0x45	sys_msgsnd	int msqid	struct msgbuf *msgp	size_t msgsz	int msgflg		
🟩	70	0x46	sys_msgrcv	int msqid	struct msgbuf *msgp	size_t msgsz	long msgtyp	int msgflg	
🟩	71	0x47	sys_msgctl	int msqid	int cmd	struct msqid_ds *buf			
🟨🟧	72	0x48	sys_fcntl	unsigned int fd	unsigned int cmd	unsigned long arg			
🟨	73	0x49	sys_flock	unsigned int fd	unsigned int cmd				
🟨	74	0x4a	sys_fsync	unsigned int fd					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	75	0x4b	sys_fdatasync	unsigned int fd					
🟩🟧	76	0x4c	sys_truncate	const char *path	long length				
🟩🟧	77	0x4d	sys_ftruncate	unsigned int fd	unsigned long length				
🟩⬛	78	0x4e	sys_getdents	unsigned int fd	struct linux_dirent *dirent	unsigned int count			
🟩	79	0x4f	sys_getcwd	char *buf	unsigned long size				
🟨	80	0x50	sys_chdir	const char *filename					
🟨	81	0x51	sys_fchdir	unsigned int fd					
🟥	82	0x52	sys_rename	const char *oldname	const char *newname				
🟥	83	0x53	sys_mkdir	const char *pathname	int mode				
🟩	84	0x54	sys_rmdir	const char *pathname					
🟥	85	0x55	sys_creat	const char *pathname	int mode				
🟥	86	0x56	sys_link	const char *oldname	const char *newname				
🟨	87	0x57	sys_unlink	const char *pathname					
🟥	88	0x58	sys_symlink	const char *oldname	const char *newname				
🟩🟩	89	0x59	sys_readlink	const char *path	char *buf	int bufsiz			

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟥	90	0x5a	sys_chmod	const char *file-name	mode_t mode				
🟥🟨	91	0x5b	sys_fchmod	unsigned int fd	mode_t mode				
🟨	92	0x5c	sys_chown	const char *file-name	uid_t user	gid_t group			
🟨	93	0x5d	sys_fchown	unsigned int fd	uid_t user	gid_t group			
🟨	94	0x5e	sys_lchown	const char *file-name	uid_t user	gid_t group			
🟨	95	0x5f	sys_umask	int mask					
🟩	96	0x60	sys_gettimeofday	struct timeval *tv	struct time-zone *tz				
🟩	97	0x61	sys_getrlimit	unsigned int re-source	struct rlimit *rlim				
🟩	98	0x62	sys_getrusage	int who	struct rusage *ru				
🟩	99	0x63	sys_sysinfo	struct sysinfo *info					
🟩	100	0x64	sys_times	struct sysinfo *info					
🟦🟨	101	0x65	sys_ptrace	long re-quest	long pid	unsigned long addr	unsigned long data		
🟩	102	0x66	sys_getuid						
🟩	103	0x67	sys_syslog	int type	char *buf	int len			
🟩	104	0x68	sys_getgid						
🟦	105	0x69	sys_setuid	uid_t uid					
🟦	106	0x6a	sys_setgid	gid_t gid					
🟩	107	0x6b	sys_geteuid						
🟩	108	0x6c	sys_getegid						
🟦🟩	109	0x6d	sys_setpgid	pid_t pid	pid_t pgid				
🟩	110	0x6e	sys_getppid						
🟩	111	0x6f	sys_getpgrp						
🟩	112	0x70	sys_setsid						

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟦🟥	113	0x71	sys_setreuid	uid_t ruid	uid_t euid				
🟦🟥	114	0x72	sys_setregid	gid_t rgid	gid_t egid				
🟩	115	0x73	sys_getgroups	int gidsetsize	gid_t *grouplist				
🟦🟩	116	0x74	sys_setgroups	int gidsetsize	gid_t *grouplist				
🟦🟨	117	0x75	sys_setresuid	uid_t *ruid	uid_t *euid	uid_t *suid			
🟩	118	0x76	sys_getresuid	uid_t *ruid	uid_t *euid	uid_t *suid			
🟦🟨	119	0x77	sys_setresgid	gid_t rgid	gid_t egid	gid_t sgid			
🟩	120	0x78	sys_getresgid	gid_t *rgid	gid_t *egid	gid_t *sgid			
🟩	121	0x79	sys_getpgid	pid_t pid					
🟦🟨	122	0x7a	sys_setfsuid	uid_t uid					
🟦🟨	123	0x7b	sys_setfsgid	gid_t gid					
🟩	124	0x7c	sys_getsid	pid_t pid					
🟩	125	0x7d	sys_capget	cap_user_header_t header	cap_user_data_t dataptr				
🟦🟨	126	0x7e	sys_capset	cap_user_header_t header	const cap_user_data_t data				
🟩	127	0x7f	sys_rt_sigpending	sigset_t *set	size_t sigsetsize				
🟩	128	0x80	sys_rt_sigtimedwait	const sigset_t *uthese	siginfo_t *uinfo	const struct timespec *uts	size_t sigsetsize		
🟦⬜	129	0x81	sys_rt_sigqueueinfo	pid_t pid	int sig	siginfo_t *uinfo			
🟨	130	0x82	sys_rt_sigsuspend	sigset_t *unewset	size_t sigsetsize				
🟩	131	0x83	sys_sigaltstack	const stack_t *ss	stack_t *oldss				
🟩	132	0x84	sys_utime	char *filename	struct utimbuf *times				

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	133	0x85	sys_mknod	const char *filename	umode_t mode	unsigned dev			
NI🟥	134	0x86	sys_uselib	NI					
🟩🌈	135	0x87	sys_personality	unsigned int personality					
🟩	136	0x88	sys_ustat	unsigned dev	struct ustat *ubuf				
🟩	137	0x89	sys_statfs	const char *pathname	struct statfs *buf				
🟩	138	0x8a	sys_fstatfs	unsigned int fd	struct statfs *buf				
🟩	139	0x8b	sys_sysfs	int option	unsigned long arg1	unsigned long arg2			
🟩	140	0x8c	sys_getpriority	int which	int who				
🟩	141	0x8d	sys_setpriority	int which	int who	int niceval			
🟦	142	0x8e	sys_sched_setparam	pid_t pid	struct sched_param *param				
🟩	143	0x8f	sys_sched_getparam	pid_t pid	struct sched_param *param				
🟩	144	0x90	sys_sched_setscheduler	pid_t pid	int policy	struct sched_param *param			
🟩	145	0x91	sys_sched_getscheduler	pid_t pid					
🟩	146	0x92	sys_sched_get_priority_max	int policy					
🟩	147	0x93	sys_sched_get_priority_min	int policy					
🟩	148	0x94	sys_sched_rr_get_interval	pid_t pid	struct timespec *interval				
🟩	149	0x95	sys_mlock	unsigned long start	size_t len				
🟩	150	0x96	sys_munlock	unsigned long start	size_t len				
🟩	151	0x97	sys_mlockall	int flags					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟩	152	0x98	sys_munlockall						
🟨	153	0x99	sys_vhangup						
🟨⬛	154	0x9a	sys_modify_ldt	int func	void *ptr	unsigned long bytecount			
🟦⬛	155	0x9b	sys_pivot_root	const char *new_root	const char *put_old				
🟦⬛	156	0x9c	sys__sysctl	struct __sysctl_args *args					
🟦🟥	157	0x9d	sys_prctl	int option	unsigned long arg2	unsigned long arg3	unsigned long arg4		unsigned long arg5
🟩	158	0x9e	sys_arch_prctl	struct task_struct *task	int code	unsigned long *addr			
🟦	159	0x9f	sys_adjtimex	struct timex *txc_p					
🟦	160	0xa0	sys_setrlimit	unsigned int resource	struct rlimit *rlim				
🟦	161	0xa1	sys_chroot	const char *filename					
🟨	162	0xa2	sys_sync						
🟦🟩	163	0xa3	sys_acct	const char *name					
🟦	164	0xa4	sys_settimeofday	struct timeval *tv	struct timezone *tz				
🟨	165	0xa5	sys_mount	char *dev_name	char *dir_name	char *type	unsigned long flags	void *data	
🟦	166	0xa6	sys_umount2	const char *target	int flags				
🟩	167	0xa7	sys_swapon	const char *specialfile	int swap_flags				
🟩	168	0xa8	sys_swapoff	const char *specialfile					
🟦	169	0xa9	sys_reboot	int magic1	int magic2	unsigned int cmd	void *arg		

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟦	170	0xaa	sys_sethostname	char *name	int len				
🟦	171	0xab	sys_setdomainname	char *name	int len				
🟦🟨	172	0xac	sys_iopl	unsigned int level	struct pt_regs *regs				
🟦🟨	173	0xad	sys_ioperm	unsigned long from	unsigned long num	int turn_on			
RM	~~174~~	~~0xae~~	~~sys_create_module~~	REMOVED IN Linux 2.6					
🟦⬛	175	0xaf	sys_init_module	void *umod	unsigned long len	const char *uargs			
🟩	176	0xb0	sys_delete_module	const chat *name_user	unsigned int flags				
RM	~~177~~	~~0xb1~~	stsys_get_kernel_syms	REMOVED IN Linux 2.6					
RM	~~178~~	~~0xb2~~	~~sys_query_module~~	REMOVED IN Linux 2.6					
🟦	179	0xb3	sys_quotactl	unsigned int cmd	const char *special	qid_t id	void *addr		
NI	180	0xb4	sys_nfsservctl	NI					
NI	181	0xb5	sys_getpmsg	NI					
NI	182	0xb6	sys_putpmsg	NI					
NI	183	0xb7	sys_afs_syscall	NI					
NI	184	0xb8	sys_tuxcall	NI					
NI	185	0xb9	sys_security	NI					
🟩	186	0xba	sys_gettid						
🟨	187	0xbb	sys_readahead	int fd	loff_t offset	size_t count			
🟩	188	0xbc	sys_setxattr	const char *path-name	const char *name	const void *value	size_t size	int flags	
🟩	189	0xbd	sys_lsetxattr	const char *path-name	const char *name	const void *value	size_t size	int flags	
🟩	190	0xbe	sys_fsetxattr	int fd	const char *name	const void *value	size_t size	int flags	

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟩	191	0xbf	sys_getxattr	const char *path-name	const char *name	void *value	size_t size		
🟩	192	0xc0	sys_lgetxattr	const char *path-name	const char *name	void *value	size_t size		
🟩	193	0xc1	sys_fgetxattr	int fd	const char *name	void *value	size_t size		
🟩	194	0xc2	sys_listxattr	const char *path-name	char *list	size_t size			
🟩	195	0xc3	sys_llistxattr	const char *path-name	char *list	size_t size			
🟩	196	0xc4	sys_flistxattr	int fd	char *list	size_t size			
🟩	197	0xc5	sys_removexattr	const char *path-name	const char *name				
🟩	198	0xc6	sys_lremovexattr	const char *path-name	const char *name				
🟩	199	0xc7	sys_fremovexattr	int fd	const char *name				
🟦⬛	200	0xc8	sys_tkill	pid_t pid	ing sig				
🟩	201	0xc9	sys_time	time_t *tloc					
🟩⬛	202	0xca	sys_futex	u32 *uaddr	int op	u32 val	struct time-spec *utime	u32 *uaddr2	u32 val3
🟦	203	0xcb	sys_sched_setaffinity	pid_t pid	unsigned int len	unsigned long *user_mask_ptr			
🟩	204	0xcc	sys_sched_getaffinity	pid_t pid	unsigned int len	unsigned long *user_mask_ptr			
NI	205	0xcd	sys_set_thread_area	NI. Use arch_prctl					
🟨⬛	206	0xce	sys_io_setup	unsigned nr_events	aio_context_t *ctxp				
🟨⬛	207	0xcf	sys_io_destroy	aio_context_t ctx					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨🟩◼	208	0xd0	sys_io_getevents	aio_context_t ctx_id	long min_nr	long nr	struct io_event *events		
🟨◼	209	0xd1	sys_io_submit	aio_context_t ctx_id	long nr	struct iocb **iocbpp			
🟨◼🟩	210	0xd2	sys_io_cancel	aio_context_t ctx_id	struct iocb *iocb	struct io_event *result			
NI	211	0xd3	sys_get_thread_area	NI. Use arch_prctl					
🟩🟩	212	0xd4	sys_lookup_dcookie	u64 cookie64	long buf	long len			
🟩🟨	213	0xd5	sys_epoll_create	int size					
NI	214	0xd6	sys_epoll_ctl_old	NI					
NI	215	0xd7	sys_epoll_wait_old	NI					
🟥	216	0xd8	sys_remap_file_pages	unsigned long start	unsigned long size	unsigned long prot	unsigned long pgoff	unsigned long flags	
🟩	217	0xd9	sys_getdents64	unsigned int fd	struct linux_dirent64 *dirent	unsigned int count			
🟩🟧	218	0xda	sys_set_tid_address	int *tidptr					
🟩◼	219	0xdb	sys_restart_syscall						
🟩🟧	220	0xdc	sys_semtimedop	int semid	struct sembuf *tsops	unsigned nsops	const struct time-spec *time-out		
🟩	221	0xdd	sys_fadvise64	int fd	loff_t offset	size_t len	int advice		
🟩	222	0xde	sys_timer_create	const clockid_t which_clock	struct sigevent *timer_event_spec	timer_t *cre-ated_timer_id			
🟩	223	0xdf	sys_timer_settime	timer_t timer_id	int flags	const struct itimer-spec *new_setting	struct itimer-spec *old_setting		
🟩	224	0xe0	sys_timer_gettime	timer_t timer_id	struct itimer-spec *setting				
🟩	225	0xe1	sys_timer_getoverrun	timer_t timer_id					
🟩	226	0xe2	sys_timer_delete	timer_t timer_id					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟩	227	0xe3	sys_clock_settime	const clockid_t which_clock	const struct timespec *tp				
🟩	228	0xe4	sys_clock_gettime	const clockid_t which_clock	struct timespec *tp				
🟩	229	0xe5	sys_clock_getres	const clockid_t which_clock	struct timespec *tp				
🟩	230	0xe6	sys_clock_nanosleep	const clockid_t which_clock	int flags	const struct timespec *rqtp	struct timespec *rmtp		
🟨🟧	231	0xe7	sys_exit_group	int error_code					
🟩🟨	232	0xe8	sys_epoll_wait	int epfd	struct epoll_event *events	int maxevents	int timeout		
🟩🟨	233	0xe9	sys_epoll_ctl	int epfd	int op	int fd	struct epoll_event *event		
🟦◼	234	0xea	sys_tgkill	pid_t tgid	pid_t pid	int sig			
🟩	235	0xeb	sys_utimes	char *filename	struct timeval *utimes				
NI	236	0xec	sys_vserver	NI					
🟦🟩	237	0xed	sys_mbind	unsigned long start	unsigned long len	unsigned long mode	unsigned long *nmask	unsigned long maxnode	unsigned flags
🟦🟩	238	0xee	sys_set_mempolicy	int mode	unsigned long *nmask	unsigned long maxnode			
🟩	239	0xef	sys_get_mempolicy	int *policy	unsigned long *nmask	unsigned long maxnode	unsigned long addr	unsigned long flags	
🟨	240	0xf0	sys_mq_open	const char *u_name	int oflag	mode_t mode	struct mq_attr *u_attr		
🟩	241	0xf1	sys_mq_unlink	const char *u_name					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	242	0xf2	sys_mq_timedsend	mqd_t mqdes	const char *u_msg_ptr	size_t msg_len	unsigned int msg_prio	const struct timespec *u_abs_timeout	
🟩	243	0xf3	sys_mq_timedreceive	mqd_t mqdes	char *u_msg_ptr	size_t msg_len	unsigned int *u_msg_prio	const struct timespec *u_abs_timeout	
🟩	244	0xf4	sys_mq_notify	mqd_t mqdes	const struct sigevent *u_notification				
🟩▨	245	0xf5	sys_mq_getsetattr	mqd_t mqdes	const struct mq_attr *u_mqstat	struct mq_attr *u_omqstat			
🟦▨	246	0xf6	sys_kexec_load	unsigned long entry	unsigned long nr_segments	struct kexec_segment *segments	unsigned long flags		
🟩	247	0xf7	sys_waitid	int which	pid_t upid	struct siginfo *infop	int options	struct rusage *ru	
🟩▨	248	0xf8	sys_add_key	const char *_type	const char *_description	const void *_payload	size_t plen		
🟩▨	249	0xf9	sys_request_key	const char *_type	const char *_description	const char *_callout_info	key_serial_t destringid		
🟩▨	250	0xfa	sys_keyctl	int option	unsigned long arg2	unsigned long arg3	unsigned long arg4	unsigned long arg5	
🟩▨	251	0xfb	sys_ioprio_set	int which	int who	int ioprio			
🟩▨	252	0xfc	sys_ioprio_get	int which	int who				
🟩	253	0xfd	sys_inotify_init						
🟨	254	0xfe	sys_inotify_add_watch	int fd	const char *pathname	u32 mask			
🟨	255	0xff	sys_inotify_rm_watch	int fd	__s32 wd				

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨🟦	256	0x100	sys_migrate_pages	pid_t pid	unsigned long maxnode	const unsigned long *old_nodes	const unsigned long *new_nodes		
🟥🟨	257	0x101	sys_openat	int dfd	const char *filename	int flags	int mode		
🟨	258	0x102	sys_mkdirat	int dfd	const char *pathname	int mode			
🟨	259	0x103	sys_mknodat	int dfd	const char *filename	int mode	unsigned dev		
🟨	260	0x104	sys_fchownat	int dfd	const char *filename	uid_t user	gid_t group	int flag	
🟩	261	0x105	sys_futimesat	int dfd	const char *filename	struct timeval *utimes			
🟩	262	0x106	sys_newfstatat	int dfd	const char *filename	struct stat *statbuf	int flag		
🟨	263	0x107	sys_unlinkat	int dfd	const char *pathname	int flag			
🟥	264	0x108	sys_renameat	int oldfd	const char *oldname	int newfd	const char *newname		
🟥	265	0x109	sys_linkat	int oldfd	const char *oldname	int newfd	const char *newname	int flags	
🟥	266	0x10a	sys_symlinkat	const char *oldname	int newfd	const char *newname			
🟩🟩	267	0x10b	sys_readlinkat	int dfd	const char *pathname	char *buf	int bufsiz		

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟥	268	0x10c	sys_fchmodat	int dfd	const char *filename	mode_t mode			
▨	269	0x10d	sys_faccessat	int dfd	const char *filename	int mode			
▨	270	0x10e	sys_pselect6	int n	fd_set *inp	fd_set *outp	fd_set *exp	struct timespec *tsp	void *sig
▨	271	0x10f	sys_ppoll	struct pollfd *ufds	unsigned int nfds	struct timespec *tsp	const sigset_t *sigmask	size_t sigsetsize	
🟩	272	0x110	sys_unshare	unsigned long unshare_flags					
🟩⬛	273	0x111	sys_set_robust_list	struct robust_list_head *head	size_t len				
▨⬛	274	0x112	sys_get_robust_list	int pid	struct robust_list_head **head_ptr	size_t *len_ptr			
🟨🟩	275	0x113	sys_splice	int fd_in	loff_t *off_in	int fd_out	loff_t *off_out	size_t len	unsigned int flags
🟨🟩	276	0x114	sys_tee	int fdin	int fdout	size_t len	unsigned int flags		
🟩	277	0x115	sys_sync_file_range	long fd	loff_t offset	loff_t bytes	long flags		
🟨	278	0x116	sys_vmsplice	int fd	const struct iovec *iov	unsigned long nr_segs	unsigned int flags		
🟩	279	0x117	sys_move_pages	pid_t pid	unsigned long nr_pages	const void **pages	const int *nodes	int *status	int flags
🟩	280	0x118	sys_utimensat	int dfd	const char *filename	struct timespec *utimes	int flags		
▨	281	0x119	sys_epoll_pwait	int epfd	struct epoll_event *events	int maxevents	int timeout	const sigset_t *sigmask	size_t sigsetsize
🟨	282	0x11a	sys_signalfd	int ufd	sigset_t *user_mask	size_t sizemask			

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	283	0x11b	sys_timerfd_create	int clockid	int flags				
🟨	284	0x11c	sys_eventfd	unsigned int count					
🟨🟦	285	0x11d	sys_fallocate	long fd	long mode	loff_t offset	loff_t len		
🟩	286	0x11e	sys_timerfd_settime	int ufd	int flags	const struct itimerspec *utmr	struct itimerspec *otmr		
🟩	287	0x11f	sys_timerfd_gettime	int ufd	struct itimerspec *otmr				
🟨	288	0x120	sys_accept4	int fd	struct sockaddr *upeer_sockaddr	int *upeer_addrlen	int flags		
🟨	289	0x121	sys_signalfd4	int ufd	sigset_t *user_mask	size_t sizemask	int flags		
🟨	290	0x122	sys_eventfd2	unsigned int count	int flags				
🟨	291	0x123	sys_epoll_create1	int flags					
🟨	292	0x124	sys_dup3	unsigned int oldfd	unsigned int newfd	int flags			
🟨	293	0x125	sys_pipe2	int *filedes	int flags				
🟨🟥	294	0x126	sys_inotify_init1	int flags					
🟨	295	0x127	sys_preadv	unsigned long fd	const struct iovec *vec	unsigned long vlen	unsigned long pos_l	unsigned long pos_h	
🟨	296	0x128	sys_pwritev	unsigned long fd	const struct iovec *vec	unsigned long vlen	unsigned long pos_l	unsigned long pos_h	
🟦⬛	297	0x129	sys_rt_tgsigqueueinfo	pid_t tgid	pid_t pid	int sig	siginfo_t *uinfo		
🟨	298	0x12a	sys_perf_event_open	struct perf_event_attr *attr_uptr	pid_t pid	int cpu	int group_fd	unsigned long flags	
🟨	299	0x12b	sys_recvmmsg	int fd	struct msghdr *mmsg	unsigned int vlen	unsigned int flags	struct timespec *timeout	

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟦🟨🟥	300	0x12c	sys_fanotify_init	unsigned int flags	unsigned int event_f_flags				
🟦🟨	301	0x12d	sys_fanotify_mark	long fanotify_fd	long flags	__u64 mask	long dfd	long pathname	
🟦	302	0x12e	sys_prlimit64	pid_t pid	unsigned int resource	const struct rlimit64 *new_rlim	struct rlimit64 *old_rlim		
🟨	303	0x12f	sys_name_to_handle_at	int dfd	const char *name	struct file_handle *handle	int *mnt_id	int flag	
🟨	304	0x130	sys_open_by_handle_at	int dfd	const char *name	struct file_handle *handle	int *mnt_id	int flags	
🟦	305	0x131	sys_clock_adjtime	clockid_t which_clock	struct timex *tx				
🟨	306	0x132	sys_syncfs	int fd					
🟨	307	0x133	sys_sendmmsg	int fd	struct mmsghdr *mmsg	unsigned int vlen	unsigned int flags		
🟩	308	0x134	sys_setns	int fd	int nstype				
🟩⬛	309	0x135	sys_getcpu	unsigned *cpup	unsigned *nodep	struct getcpu_cache *unused			
🟨🟦	310	0x136	sys_process_vm_readv	pid_t pid	const struct iovec *lvec	unsigned long liovcnt	const struct iovec *rvec	unsigned long riovcnt	unsigned long flags
🟨🟦	311	0x137	sys_process_vm_writev	pid_t pid	const struct iovec *lvec	unsigned long liovcnt	const struct iovcc *rvec	unsigned long riovcnt	unsigned long flags
🟩⬛	312	0x138	sys_kcmp	pid_t pid1	pid_t pid2	int type	unsigned long idx1	unsigned long idx2	
🟦⬛	313	0x139	sys_finit_module	int fd	const char __user *uargs	int flags			
🟩	314	0x13a	sys_sched_setattr	pid_t pid	struct sched_attr __user *attr	unsigned int flags			

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟩	315	0x13b	sys_sched_getattr	pid_t pid	struct sched_attr __user *attr	unsigned int size	unsigned int flags		
🟥	316	0x13c	sys_renameat2	int olddfd	const char __user *oldname	int newdfd	const char __user *newname	unsigned int flags	
🟦	317	0x13d	sys_seccomp	unsigned int op	unsigned int flags	const char __user *uargs			
🟩	318	0x13e	sys_getrandom	char __user *buf	size_t count	unsigned int flags			
🟨	319	0x13f	sys_memfd_create	const char __user *uname_ptr	unsigned int flags				
🟦⬛	320	0x140	sys_kexec_file_load	int kernel_fd	int initrd_fd	unsigned long cmdline_len	const char __user *cmdline_ptr	unsigned long flags	
🟩	321	0x141	sys_bpf	int cmd	union bpf_attr *attr	unsigned int size			
🟥	322	0x142	sys_execveat	int dfd	const char __user *filename	const char __user *const __user *argv	const char __user *const __user *envp	int flags	
🟨⬛	323	0x143	sys_userfaultfd	int flags					
🟩	324	0x144	sys_membarrier	int cmd	int flags				
🟩	325	0x145	sys_mlock2	unsigned long start	size_t len	int flags			
🟨	326	0x146	sys_copy_file_range	int fd_in	loff_t __user *off_in	int fd_out	loff_t __user * off_out	size_t len	unsigned int flags
🟨	327	0x147	sys_preadv2	unsigned long fd	const struct iovec __user *vec	unsigned long vlen	unsigned long pos_l	unsigned long pos_h	int flags

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	328	0x148	sys_pwritev2	unsigned long fd	const struct iovec __user *vec	unsigned long vlen	unsigned long pos_l	unsigned long pos_h	int flags
🟥	329	0x149	sys_pkey_mprotect	unsigned long start	size_t len	unsigned long prot	int pkey		
🟥	330	0x14a	sys_pkey_alloc	unsigned long flags	unsigned long init_access_right				
🟩	331	0x41b	sys_pkey_free	int pkey					
🟩	332	0x41a	sys_statx	int dfd	const char *filename	unsigned flags	unsigned int mask	struct statx *buffer	
🟨🟩⬛	333	0x41b	sys_io_pgetevents	aio_context_t ctx_id	long min_nr	long nr	struct *events	struct *timeout	struct *sig
🟩	334	0x41c	sys_rseq	struct rseq __user *rseq	uint32_t rseq_len	int flags	uint32_t sig		
	...	...							
	...	...							
🟦	424	0x1a8	sys_pidfd_send_signal	int pidfd	int sig	siginfo_t __user *info	unsigned int flags		
🟩	425	0x1a9	sys_io_uring_setup	u32 entries	struct io_uring_params __user *p				
🟩	426	0x1aa	sys_io_uring_enter	unsigned int fd	u32 to_submit	u32 min_complete	u32 flags	const sigset_t __user *sig	size_t sigsz
🟩	427	0x1ab	sys_io_uring_register	unsigned int fd	unsigned int op	void __user *arg	unsigned int nr_args		
🟦⬛	428	0x1ac	sys_open_tree	int dfd	const char __user *path	unsigned flags			
🟦⬛	429	0x1ad	sys_move_mount	int from_dfd	const char __user *from_path	int to_dfd	const char __user *to_path	unsigned int ms_flags	
🟦⬛	430	0x1ae	sys_fsopen	const char __user *fs_name	unsigned int flags				

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟦▨	431	0x1af	sys_fsconfig	int fs_fd	unsigned int cmd	const char __user *key	const void __user *value	int aux	
🟦▨	432	0x1b0	sys_fsmount	int fs_fd	unsigned int flags	unsigned int ms_flags			
🟦▨	433	0x1b1	sys_fspick	int dfd	const char __user *path	unsigned int flags			
🟩	434	0x1b2	sys_pidfd_open	pid_t pid	unsigned int flags				
🟥	435	0x1b3	sys_clone3	struct clone_args __user *uargs	size_t size				
	...	...							
	...	...							
🟨	512	0x200	sys_rt_sigaction	*32-bit: see 64-bit version*					
🟩	513	0x201	sys_rt_sigreturn	*32-bit: see 64-bit version*					
🟥	514	0x202	sys_ioctl	*32-bit: see 64-bit version*					
🟨	515	0x203	sys_readv	*32-bit: see 64-bit version*					
🟨	516	0x204	sys_writev	*32-bit: see 64-bit version*					
🟨	517	0x205	sys_recvfrom	*32-bit: see 64-bit version*					
🟨	518	0x206	sys_sendmsg	*32-bit: see 64-bit version*					
🟨	519	0x207	sys_recvmsg	*32-bit: see 64-bit version*					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟥	520	0x208	sys_execve	*32-bit: see 64-bit version*					
🟦🟨	521	0x209	sys_ptrace	*32-bit: see 64-bit version*					
🟩	522	0x20a	sys_rt_sigpending	*32-bit: see 64-bit version*					
🟩	523	0x20b	sys_rt_sigtimedwait	*32-bit: see 64-bit version*					
🟦◼	524	0x20c	sys_rt_sigqueueinfo	*32-bit: see 64-bit version*					
🟩	525	0x20d	sys_sigaltstack	*32-bit: see 64-bit version*					
🟩	526	0x20e	sys_timer_create	*32-bit: see 64-bit version*					
🟩	527	0x20f	sys_mq_notify	*32-bit: see 64-bit version*					
🟦◼	528	0x210	sys_kexec_load	*32-bit: see 64-bit version*					
🟩	529	0x211	sys_waitid	*32-bit: see 64-bit version*					
🟩◼	530	0x212	sys_set_robust_list	*32-bit: see 64-bit version*					
🟩◼	531	0x213	sys_get_robust_list	*32-bit: see 64-bit version*					
🟨	532	0x214	sys_vmsplice	*32-bit: see 64-bit version*					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	533	0x215	sys_move_pages	*32-bit: see 64-bit version*					
🟨	534	0x216	sys_preadv	*32-bit: see 64-bit version*					
🟨	535	0x217	sys_pwritev	*32-bit: see 64-bit version*					
🟦▨	536	0x218	sys_rt_tgsigqueueinfo	*32-bit: see 64-bit version*					
🟨	537	0x219	sys_recvmmsg	*32-bit: see 64-bit version*					
🟨	538	0x21a	sys_sendmmsg	*32-bit: see 64-bit version*					
🟨	539	0x21b	sys_process_vm_readv	*32-bit: see 64-bit version*					
🟨	540	0x21c	sys_process_vm_writev	*32-bit: see 64-bit version*					
🟨	541	0x21d	sys_setsockopt	*32-bit: see 64-bit version*					
🟩	542	0x21e	sys_getsockopt	*32-bit: see 64-bit version*					
🟨🟥▨	543	0x21f	sys_io_setup	*32-bit: see 64-bit version*					
🟨▨	544	0x220	sys_io_submit	*32-bit: see 64-bit version*					
🟥	545	0x221	sys_execveat	*32-bit: see 64-bit version*					

cc	%rax	%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
🟨	546	0x222	sys_preadv2	*32-bit: see 64-bit version*					
🟨	547	0x223	sys_pwritev2	*32-bit: see 64-bit version*					

Critical: 🟥
Needs privilege: 🟦
To chain: 🟨
Safe: 🟩
No state change: 🟩
Other: 🟧
No wrapper: ⬛

H Security-sensitive System Calls

H.1 Security-sensitive system calls in *in-the-wild*

rax	hex	System call	rdi	rsi	rdx	r10	r8	r9
009	0x009	sys_mmap	addr	len	prot	flags	fd	off
010	0x00a	sys_mprotect	start	len	prot			
059	0x03b	sys_execve	*filename	**argv	**envp			
322	0x142	sys_execveat	dfd	*filename	**argv	**envp	flags	
329	0x149	sys_pkey_mprotect	start	len	prot	pkey		
520	0x208	sys_execve_32	*filename	**argv	**envp			
545	0x221	sys_execveat_32	dfd	*filename	**argv	**envp	flags	

H.2 Security-sensitive system calls in *in-audit*

rax	hex	System call	rdi	rsi	rdx	r10	r8	r9
002	0x002	sys_open	*filename	flags	mode			
009	0x009	sys_mmap	addr	len	prot	flags	fd	off
010	0x00a	sys_mprotect	start	len	prot			
013	0x00d	sys_rt_sigaction	sig	*act	*oact	sigsetsize		
016	0x010	sys_ioctl	fd	cmd	arg			
030	0x01e	sys_shmat	shmid	*shmaddr	shmflg			
041	0x029	sys_socket	family	type	protocol			
053	0x035	sys_socketpair	family	type	protocol	*usockvec		
056	0x038	sys_clone	clone_flags	newsp	*parent_tid	*child_tid		
059	0x03b	sys_execve	*filename	**argv	**envp			
082	0x052	sys_rename	*oldname	*newname				
083	0x053	sys_mkdir	*pathname	mode				
085	0x055	sys_creat	*pathname	mode				
086	0x056	sys_link	*oldname	*newname				
088	0x058	sys_symlink	*oldname	*newname				
090	0x05a	sys_chmod	*filename	mode				
091	0x05b	sys_fchmod	fd	mode				
216	0x0d8	sys_remap_file_pages	start	size	prot	pgoff	flags	
257	0x101	sys_openat	dfd	*filename	flags	mode		
264	0x108	sys_renameat	oldfd	*oldname	newfd	*newname		
265	0x109	sys_linkat	oldfd	*oldname	newfd	*newname	flags	
266	0x10a	sys_symlinkat	*oldname	newfd	*newname			
268	0x10c	sys_fchmodat	dfd	*filename	mode			
294	0x126	sys_inotify_init1	flags					
316	0x13c	sys_renameat2	olddfd	*oldname	newdfd	*newname	flags	
322	0x142	sys_execveat	dfd	*filename	**argv	**envp	flags	
329	0x149	sys_pkey_mprotect	start	len	prot	pkey		
330	0x14a	sys_pkey_alloc	flags	init_access_right				
435	0x1b3	sys_clone3	*uargs	size				
514	0x202	sys_ioctl_32	fd	cmd	arg			
520	0x208	sys_execve_32	*filename	**argv	**envp			
545	0x221	sys_execveat_32	dfd	*filename	**argv	**envp	flags	