'Weird Machine' patterns

Article · March 2014

DOI: 10.1007/978-3-319-04447-7_13

6 authors, including:

Sergey Bratus
Dartmouth College

Daniel Bilar
Norwich University

Chapter 13
'Weird Machine' Patterns

Sergey Bratus, Julian Bangert, Alexandar Gabrovsky, Anna Shubina, Michael Locasto and Daniel Bilar

Abstract You do not understand how your program *really* works until it has been exploited. We believe that computer scientists and software engineers should regard the activity of modern exploitation as an applied discipline that studies both the actual computational properties and the practical computational limits of a target platform or system. Exploit developers study the computational properties of software that are not studied elsewhere, and they apply unique engineering techniques to the challenging engineering problem of dynamically patching and controlling a running system. These techniques leverage software and hardware composition mechanisms in unexpected ways to achieve such control. Although unexpected, such composition is not arbitrary, and it forms the basis of a coherent engineering workflow. This chapter contains a top-level overview of these approaches and their historical development.

1 Introduction

When academic researchers study exploitation, they mostly concentrate on two classes of attack-related artifacts: "malicious code" (malware, worms, shellcode) and, lately, "malicious computation" (exploits via crafted data payloads containing no native code, a popular exploiter technique. These techniques have been discussed

S. Bratus (✉) · J. Bangert · A. Gabrovsky · A. Shubina
Dartmouth College, Hanover, New Hampshire, USA
e-mail: sergey@cs.dartmouth.edu
http://www.cs.dartmouth.edu/ sergey/

M. E. Locasto
University of Calgary, Alberta, Canada
http://pages.cpsc.ucalgary.ca/locasto/

D. Bilar
Siege Technologies, Manchester, New Hampshire, USA
http://www.siegetechnologies.com/

C. Blackwell and H. Zhu (eds.), *Cyberpatterns*, DOI: 10.1007/978-3-319-04447-7_13,

in the hacker community since before early 2000s ([1] offers a brief history sketch). They were brought a decade later to the attention of academia by Shacham [2, 3] in 2007. These artifacts are tackled with a variety of approaches, from machine learning on either captured payloads or execution traces to automatic construction of exploit payloads for specific targets. The goals of such studies also vary, from straightforward detection of particular payloads in network traffic to finding and removing vulnerabilities or asserting that a vulnerability is not exploitable beyond a crash or denial-of-service. Considerable technical depth has been reached in all of these directions, yet we seem no closer to the ultimate goal of constructing trustworthy, non-exploitable software.

We argue that focusing on just these two classes of artifacts is a limiting factor in our understanding of exploitation (and therefore of how to prevent it). We believe that, as a means of making progress toward the goal of more fundamentally secure software, we must understand how exploitation relates to *composition*, which is fundamental to all modern software construction. Understanding the patterns of this relationship will expose new artifacts to study and indicate new technical directions.

We note that practical exploitation has long been about *composing* the attacker computation with the native computation of the target, allowing most of the target's functions to proceed normally, without undue interference. We posit that such composition is the source of the most powerful and productive concepts and methodologies that emerge from exploitation practice.

When researchers focus on attack artifacts alone, they frequently miss an important point of successful exploitation: the exploited system needs to remain available and reliably usable for the attacker.

In order to support this assertion and further discuss exploitation, we need to make an important terminological point. The word *hacking* is used to refer to all kinds of attacks on computer systems, including those that merely shut down systems or otherwise prevent access to them (essentially achieving nothing that could not be achieved by cutting a computer cable). Many activities labeled as "hacking" lack sophistication. In this chapter we focus on *exploitation* or, more precisely, *exploit programming*. We take *exploitation* and *exploit programming* to mean subverting the system to make it work for the attacker—that is, lend itself to being programmed by the attacker. Exploiters are less interested in causing BSODs, kernel panics, and plain network DOS attacks that merely result in a DoS on the target and cannot otherwise be leveraged and refined to take control over the system rather than disabling it.

Not surprisingly, preventing a disabling crash and subsequently "patching up" the target into a stable running state requires significantly more expertise and effort than, say, a memory-corrupting DoS. By achieving this state exploiters demonstrate a sophisticated understanding of the target platform,[1] often beyond the ken of its developers or even system programmers.

In this chapter we review a series of classic exploitation techniques from the perspective of composition. Many of these techniques have been extensively described

[1] It also serves as an excellent teaching aid in advanced OS courses; see, e.g., [4].

and reviewed from other perspectives; however, their compositional aspect is still treated as ad hoc, and has not, as far as we know, been the subject of systematic analysis. Specifically, we regard composition as the basic unit of activity in an engineering workflow, whether that workflow is a traditional software engineering workflow or a workflow focused on engineering an exploit. We compare these workflows in the Sect. 2.

Since our focus is on composition, we do not distinguish between techniques used by *rootkits* vs. *exploits*. Rootkits are traditionally separated from other exploit-related artifacts such as exploits proper, "shellcode", etc., since they are meant to be installed by the successful attacker who already attained the "root" level of privilege by other means. However, we note that such installation often involves means of composition that are only available to developers, not administrators however privileged; thus, composing parts of a rootkit with the system poses challenges due to lacking information and limited available context. The complexity of such challenges may vary, but they have the same nature as those faced by an exploit programmer, and indeed similar techniques are used to overcome them. In our discussion, we draw equally upon rootkit and exploit examples.

We posit that composition-centric analysis is required for designing defensible systems (see Sect. 4). The practical properties of composition in actual computer systems uncovered and distilled by hacker research have often surprised both designers and defenders. We believe that the relevant methods here must be cataloged and generalized to help approach the goal of *secure composition* in future designs.

2 A Tale of Two Engineering Workflows

"Language design is library design."
– B. Stroustrup

Hacking, vulnerability analysis, and exploit programming are generally perceived to be difficult and arcane activities. The development of exploits is still seen as something unrepeatable and enabled only by some unfortunate and unlikely combination of events or conditions. Almost by definition, something as imbued with arbitrary chance cannot or should not be an engineering discipline or workflow. Popular perception casts these activities as requiring specialized cross–layer knowledge of systems and a talent for "crafting" input.

This chapter asserts that what seems arcane is really only unfamiliar. In fact, although it may be difficult to conceive of exploit development as anything other than fortunate mysticism, we argue that its structure is exactly that of a software engineering workflow. The difference emerges in the specific constructs at each stage, but the overall activities remain the same. A software developer engineers in terms of sequences of function calls operating on abstract data types, whereas an exploit developer engineers in terms of sequences of machine–level memory reads and writes. The first one programs the system in terms of what its compile-time API

318981_1_en_13_chapter ☑ TYPESET ☐ DISK ☐ LE ☑ CP Disp.:24/3/2014 Pages: 15 Layout: T1-Standard

promises; the other programs it in terms of what its runtime environment actually contains.

This section contains a brief comparison of these two engineering workflows. We do so to help give a conceptual frame of reference to the enumeration of exploit techniques and composition patterns detailed in Sect. 3.

The main difference between the two workflows is that the exploit engineer must first recover or understand the semantics of the runtime environment. In either case, programming is composition of functionality.

In the "normal" workflow of software engineering, the programmer composes familiar, widely-used libraries, primitive language statements (repetition and decision control structures), and function calls to kick input data along a processing path and eventually produce the result dictated by a set of functional requirements.

In the exploit workflow, the reverser or exploit engineer attempts to build this programming toolkit from scratch: the languages and libraries that the software engineer takes for granted are not of *direct* use to the exploit developer. Instead, these elements define a landscape from which the exploit developer must compose and create his own toolkit, language primitives, and component groups. The first job of the vulnerability analyst or reverse engineer is therefore to understand the latent functionality existing in runtime environments that the software engineer either neglects or does not understand.

2.1 The Software Engineer

Based on functional requirements, a software engineer's goal is to cause some expected functionality happen. In essence, this kind of programming is the task of choosing a sequence of library calls and composing them with language primitives like decision control structure and looping control structures. Data structures are created to capture the relevant properties of the system's input; this structure usually dictates how processing (i.e., control flow) occurs.

A software engineer follows roughly this workflow path:

1. design and specify data types
2. design data flow relationships (i.e., an API)
3. write down source code implementing the data types and API
4. ask compiler and assembler to translate code
5. ask OS to load binary, invoke the dynamic linker, and create memory regions
6. run program according to the control flow as conceived in the source level.

In this workflow, we can see the software engineer engaged in: memory layout, specifying control flow, program construction, program delivery (loading) and translation, and program execution. As we will see below, the exploit engineer engages in much the same set of tasks.

The software engineer's goal is to bring order to a composition of procedures via compilation and assembly of machine code. One does this through tool chains, design patterns, IDEs, and popular languages—the software engineer therefore does not need to relearn the (public) semantics of these operations every time he prepares to program.

These conventions are purely an effort–saving device aimed at increasing productivity by increasing the lines of code and features implemented in them. These patterns, tools, and aids reduce the level of thought required to emit a sequence of function calls that satisfy the functional requirements. They are an effort to deal with complexity. The goal of software engineers in dealing with complexity is to eliminate or hide it.

2.2 The Exploit Engineer

In contrast, exploit engineers also deal with complexity, but their goal is to manipulate it—expressiveness, side effects, and implicit functionality are a collective boon, not a bane. Any operations an exploit engineer can get "for free" increase his exploit toolkit, language, or architecture. A software engineer attempts to hide or ignore side effects and implicit state changes, but the very things encouraged by traditional engineering techniques like "information hiding" and encapsulation on the other side of an API become recoverable primitives for a reverser or exploit engineer.

The main difference in the workflows is the preliminary step: you have to learn on a case by case or scenario by scenario basis what "language" or computational model you should be speaking in order to actually begin programming toward a specific functional end. Based on some initial access, the first goal is to understand the system enough to recover structure of "programming" primitives. The workflow is thus:

1. identify system input points
2. recapture or expose trust relationships between components (functions, control flow points, modules, subroutines, etc.)
3. recover the sequencing composition of data transformations (enumerate layer crossings)
4. enumerate instruction sequences / primitives / gadgets
5. program the process address space (prepare the memory image and structure)
6. deliver the exploit.

In this workflow, we can see the exploit engineer engaged in: recovering memory layout, specifying control flow, program construction, program delivery (loading) and translation, and program execution. We note that these steps may not (and need not be) sperabale: Unlike the software engineering workflow, the delivery of an exploit (i.e., loading a program) can be mixed up and interposed with translation of the program and preparation of the target memory space. Even though these activities might be more tightly coupled for an exploit developer, much of the same discipline remains.

Recent academic advances have the potential of automating (at least partially) the preparatory steps (1–4) the exploiter's workflow. Holler's *LangFuzz* tool automates black-box fuzz testing of context-free grammar engines. It generates test cases from a given context-free grammar to exposes via fault generation inter-components' trust relations [5]. Caballero proposed and implemented the *Dispatcher* tool for automatic protocol reverse-engineering given an undocumented protocol or le format. Thus includes the structure of all messages that comprise the protocol in addition to the protocol state machine, which captures the sequences of messages that represent valid sessions of the protocol. As a proof of concept, his group managed to extract the grammar of Mega-D (a spam botnet), which sported an undocumented, encrypted Command & Control protocol [6]. The output of these tools can be repurposed for defenses. Samuels proposed a simple but clever approach against certain type confusion attacks through a generalizable annotated parse-tree-grammar scheme. Such annotated grammars can be converted to push-down automata from which input stress test can be derived [7]. The urgent need for such defenses is demonstrated by Shmatikov and Wang analysis of and attacks against AV parsers and undefined behavior in C language compilers, respectively [8, 9].

One major challenge exists for the exploit engineer: recovering the unknown unknowns. Although they can observe side effects of mainline execution or even slightly fuzzed execution, can they discover the side effects of "normally" dormant or latent "normal" functionality (e.g., an internationalization module that is never invoked during normal operation, or configuration code that has only been invoked in the "ancient past" of this running system)? This challenge is in some sense like the challenge a software engineer faces when exploring a very large language library (e.g., the Java class library API).

2.3 A Simple Example of Exploit Programming

Before we turn our attention to reviewing the composition patterns of hacking, we give a brief example of constructing an exploit as a programming task, to show the kind of workflow involved. The reader already familiar with such concept may skip directly to Sect. 3.

Assume a Program P that reads the contents of a file F into a buffer located on the stack and displays this content to standard output.

Note that our point here is not to say that stack-based buffer overflows are of independent or modern interest; rather, we use this scenario as the simplest illustration of exploitation as programming, where most readers likely already have a conception of some of the issues in play, from both the software developer side (why this vulnerability exists: failure to check lengths) and the exploit engineer side (where to inject and how to structure shellcode)—it is popularly "understood" enough.

The simplest possible core task in the exploit engineer's workflow in this scenario is to map input data to memory space. Concretely: where will the bytes that overwrite the return address and saved ebp land? And what should the content of my new return address be (i.e., the address of the file's content on the stack)?

318981_1_en_13_chapter ☑ TYPESET ☐ DISK ☐ LE ☑ CP Disp.:24/3/2014 Pages: 15 Layout: T1-Standard

In this scenario, file F is simultaneously:

- bytes in persistent storage
- bytes traversing kernel space via the read (2) implementation (and callchain)
- data for the program (i.e., contents of a buffer)
- an overlay of some part of the process address space
- bytecode for an automaton implicitly embedded in the target program (the so-called "weird machine", see Sect. 3) that will carry out malicious computation
- shellcode for the CPU.

Understanding the layout of this memory is vital to actually constructing an input file (i.e., bytecode) to program the exploit machine. In addition to understanding the memory layout as a living artifact generated by both the compiler and the runtime system (i.e., how the OS sets up the process address space and memory regions), an exploit engineer must also understand other implicit operators, transformers, and parsers co-existing in the *real* computational machine. He must understand where they interpose on the processing path, how they are invoked, and what *actual* set of transformations they have on the data (i.e., bytecode) as it journeys through the system toward various resting places. He must ask: is my data filtered? Does it have a simple transformation or encoding applied to it (e.g., Base64 encoding, `toupper()`)? Do certain special bytes (i.e., NULL) truncate the data? Is the data copied to new locations? Is the data reshuffled or reorganized internally?

When considering more complex examples, an exploit engineer must map and understand other memory regions, including the heap and its management data structures, dynamic data embedded within various memory regions, and threading (concurrency). Such things now present additional computational primitives to place in context and understand. In some sense, the sum total composition of all these mechanisms is a compiler, translator, or interpreter for your bytecode—and you must first understand how that compiler works. When you do, you can undertake the process of writing and constructing bytecode appropriate to accomplishing your goal, whether that is to drop shell, open a port, install a rootkit, exfiltrate a file, etc.

3 Patterns

3.1 Exploitation as Programming "Weird Machines"

Bratus et al. [1] summarized a long-standing hacker intuition of exploits as *programs, expressed as crafted inputs, for execution environments implicitly present in the target as a result of bugs or unforeseen combination of features* ("weird machines"), which are reliably driven by the crafted inputs to perform unexpected computations. More formally, the crafted inputs that constitute the exploit drive an input-accepting automaton already implicitly present in the target's input-handling implementation, its sets of states and transitions owing to the target's features, bugs or combinations thereof.

The implicit automaton is immersed into or is part of the target's execution environment; its processing of crafted input is part of the "malicious computation" — typically, the part that creates the initial compromise, after which the exploiter can program the target with more conventional means. The crafted input is both a program for that automaton and a constructive proof of its existence. Further discussion from the practical exploit programming standpoint can be found in Dullien [10], from a theory standpoint in Sassaman [11].

This perspective on exploit programming considers the exploit target as harboring a virtual computing architecture, to which the input data serve as *bytecode*, similar to, say, how compiled Java bytecode drives the Java virtual machine. In other words, the target's input is viewed as an actual program, similar to how the contents of a Turing machine's tape can be considered a program. Thus what is liable to be seen by developers as "inert data" such as inputs or metadata is in fact conceptually promoted to a vehicle of programming the target; in a sense, the exploiter treats the data as running and acting on the target program, not the other way around. Further discussion of this can be found in Shapiro [12].

In the following items, we focus on one critical aspect of the implicit exploit execution environments and the computations effected in them by exploit-programs: they must reliably co-exist with the native, intended computations both for their duration and in their effects, while their composition is done in contexts more limited and lacking critical information as compared to the system's intended scenarios. This is far from trivial on systems where state that is "borrowed" by the exploit computation's thread of control is simultaneously used by others. It involves dissecting and "slimming down" interfaces to their actual implementation primitives and finding out unintended yet stable properties of these primitives.

3.2 Recovering Context, Symbols, and Structure

To compose its computation with a target, an exploit must refer to the objects it requires in its virtual address space (or in other namespaces). In essence, except in the most trivial cases, a "name service" of a kind (ranging from ad-hoc to the system's own) is involved to reconstruct the missing information.

Early exploits and rootkit install scripts relied on hard-coded fixed addresses of objects they targeted, since back then memory virtual space layouts were identical for large classes of targets.[2] As targets' diversity increased, naturally or artificially (e.g., OpenWall, PaX, other ASLR), exploits progressed to elaborate address space layout reconstruction schemes and co-opting the system's own dynamic linking and/or trapping debugging.

[2] This fact was not well understood by most engineers or academics, who regarded below-compiler OS levels as unpredictable; Stephanie Forrest deserves credit for putting this and other misconceptions into broader scientific perspective.

Cesare [13] describes the basic mechanism behind ELF linking—based on little more that careful reading of the ELF standard. However, it broke the opacity and resulted in an effective exploit technique, developed by others, e.g., [14]. In [15] mayhem builds on the same idea by looking into the significance and priority of ELF's `.dynamic` symbols. Nergal [16] co-opted Linux's own dynamic linker into an ROP[3] crafted stack frame-chaining scheme, to have necessary symbols resolved and libraries loaded. Oakley [17] showed how to co-opt the DWARF-based exception handling mechanism.

Skape [18] takes the understanding of ELF in a different direction by showing how its relocation mechanism works and how that could be used for unpacking obfuscated Windows binaries. Recent work by Shapiro [12] demonstrated that the relocation metadata in ELF binaries is actually enough to drive Turing-complete computations on the Linux dynamic linker-loader. The ABI metadata in these examples serves as "weird machine" bytecode for the Turing machine implicitly embedded in the RTLD code.

In all of the above detailed understanding of a mechanism comes before the insight of how an exploit could be built; in fact, once the mechanism is clear at the "weird machine" level, its exploitation use is almost an afterthought.

3.3 Preparing Vulnerable System State

Earlier classes of exploits leveraged conditions and configurations (such as memory allocation of relevant objects) present in the target's state through all or most runs. Subsequent advancements such as Sotirov's [19] demonstrated that *otherwise non-exploitable targets can have their state carefully prepared by way of a calculated sequence of requests and inputs for an exploitable configuration to be instantiated.*

This pattern of pre-compositional state-construction of targets is becoming essential, as protective entropy-injecting techniques prevent setting up an effective "name service". Recent examples [20, 21] show its applications to modern heaps (the former for the Windows low fragmentation heap), in presence of ASLR and DEP. Moreover, this method can target the injected entropy *directly*, by bleeding it from the target's state (e.g., [22]).

3.4 Piercing Abstraction

Developers make use of abstractions to decrease implementation effort and increase code maintainability. However, abstractions hide the details of their implementation and as they become part of a programmers daily vocabulary, the implementation

[3] Which it pre-dates, together with other hacker descriptions of the technique, by five to seven years.

318981_1_en_13_chapter ☑ TYPESET ☐ DISK ☐ LE ☑ CP Disp.:24/3/2014 Pages: 15 Layout: T1-Standard

details are mostly forgotten. For example, few programmers worry about how a function call is implemented at the machine level or how the linking and loading mechanisms assign addresses to imported symbols.

Exploit engineers, however, distill abstractions into their implementation primitives and synthesize new composition patterns from them. Good examples of this are found in [16], who modifies the return addresses on the stack to compose existing code elements into an exploit, and the `LOCREATE` [18] packer which obfuscates binary code by using the primitives for dynamic linking.

3.5 Balancing Context Constraints

> Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication.
>
> A. Perlis

When a software architect considers how much context to pass through an interface, he has to balance competing constraints (see Sect. 4.2 in [23] for discussion of etiology and formalization sketch). Either a lot of context is passed, reducing the flexibility of the code, or too little context is preserved and the remaining data can no longer be efficiently validated by code operating on it, so more assumptions about the input have to be trusted. Exploiters explore this gap in assumptions, and distill the unintended side-effects to obtain *primitives*, from which weird machines are constructed [10, 24, 25]. We posit that understanding this gap is the way to more secure API design.

3.6 Bit Path Tracing of Cross-Layer Flows

When an exploiter studies a system, he starts with bit-level description of its contents and communications. Academic textbooks and user handbooks, however, typically do not descend to bit level and provide only a high-level description of how the system works. A crucial part of such bit-level description is the flow of bits between the conceptual design layers of the system: i.e. a binary representation of the data and control flow between layers.

Constructing these descriptions may be called the cornerstone of the hacker methodology. It precedes the search for actual vulnerabilities and may be thought of as the modeling step for constructing the exploit computation. The model may ignore large parts of the target platform but is likely to punctiliously describe the minutiae of composition mechanisms that actually tie the implementations of the layers together.

For example, the AlephOne Phrack article [26] famous for its description of stack buffer overflows also contained a bit-level description of UNIX system calls, which for many readers was in fact their first introduction to syscall mechanisms. Similarly, other shellcode tutorials detailed the data flow mechanisms of the target's ABIs (such as various calling conventions and the structure of libraries). In networking, particular attention was given to wrapping and unwrapping of packet payloads at each level of the OSI stack model, and libraries such as libnet and libdnet were provided for emulating the respective functionality throughout the stack layers.

What unites the above examples is that in all of them exploiters start analyzing the system by tracing the flow of bits within the target and enumerating the code units that implement or interact with that flow. The immediate benefits of this analysis are at least two-fold: locating of less known private or hidden APIs and collecting potential exploitation primitives or "cogs" of "weird machines", i.e. code fragments on which crafted data bits act in predictable way.

Regardless of its immediate benefits, though, bit-level cross-layer flow descriptions also provide useful structural descriptions of the system's architecture, or, more precisely, of the mechanisms that underly the structure, such as the library and loadable kernel functionality, DDKs, and network stack composition.

For instance, the following sequence of Phrack articles on Linux rootkits is a great example of deep yet concise coverage of the layers in the Linux kernel architecture: *Sub proc_root Quando Sumus (Advances in Kernel Hacking)* [27] (VFS structures and their linking and hijacking), *5 Short Stories about execve (Advances in Kernel Hacking II)* [28] (driver/DDK interfaces, different binary format support), and *Execution path analysis: finding kernel based rootkits* [29] (instrumentation for path tracing). Notably, these articles at the cusp where three major UNIX innovations meet: VFS, kernel state reporting through pseudo-filesystems (e.g., `/proc`), and support for different execution domains/ABI. These articles described the control and data flows through a UNIX kernel's component layers and their interfaces in great detail well before tools like DTrace and KProbes/SystemTap brought tracing of such flows within common reach.

It is worth noting that the ELF structure of the kernel binary image, the corresponding structure of the kernel runtime, and their uses for reliably injecting code into a running kernel (via writing `/dev/kmem` or via some kernel memory corruption primitive). In 1998, the influential *Runtime kernel kmem patching* [30] made the point that even though a kernel may be compiled without loadable kernel module support, it still is a structured runtime derived from an ELF image file, in which symbols can be easily recovered, and the linking functionality can be provided without difficulty by a minimal userland "linker" as long as it has access to kernel memory. Subsequently, mature kernel function hooking frameworks were developed (e.g., *IA32 Advanced function hooking* [31]).

Dynamic linking and loading of libraries (shared binary objects) provide another example. This is a prime example of composition, implicitly relied upon by every modern OS programmer and user, with several supporting engineering mechanisms and abstractions (ABI, dynamic symbols, calling conventions). Yet, few resources exist that describe this key mechanism of interposing computation; in fact, for a long time hacker publications have been the best resource for understanding the underlying binary data structures (e.g., *Backdooring binary objects* [32]), the control flow of dynamic linking (e.g., *Cheating the ELF* [33] and *Understanding Linux ELF*

RTLD internals [34]), and the use of these structures for either binary infection (e.g., the original *Unix ELF parasites and virus*) or protection (e.g., *Armouring the ELF: Binary encryption on the UNIX platform* [35]).

A similar corpus of articles describing the bit paths and layer interfaces exists for the network stacks. For the Linux kernel stack, the *Netfilter* architecture represents a culmination of this analysis. By exposing and focusing on specific hooks (tables, chains), Netfilter presents a clear and concise model of a packet's path through the kernel; due to this clarity it became both the basis of the Linux's firewall and a long series of security tools.

Not surprisingly, exploitative modifications of network stacks follow the same pattern as other systems rootkits. *Passive Covert Channels Implementation in Linux Kernel* [36] is a perfect example: it starts with describing the interfaces traversed on a packet's path through the kernel (following the Netfilter architecture), and then points out the places where a custom protocol handler can be inserted into that control flow, using the stack's native protocol handler interfaces.

3.7 Trap-Based Programming and Composition

In application programming, traps and exceptions are typically not treated as "first-class" programming primitives. Despite using powerful exception-handling subsystems (such as GCC's *DWARF*-based one, which employs Turing-complete bytecode), applications are not expected to perform much of their computation in traps or exceptions and secondary to the main program flow. Although traps are obviously crucial to systems programming, even there the system is expected to exit their handlers quickly, performing as little and as simple computation as possible, for both performance and context management reasons.

In exploit programming and reverse engineering (RE), traps are the *first-class programming primitives*, and trap handler overloading is a frequently used technique. The target platform's trap interfaces, data structures, and contexts are carefully studied, described, and modeled, then used for reliably composing an exploit or a comprehension computation (i.e., a specialized tracer of debugger) with the target.

The tracing and debugging subsystems in OS kernels have long been the focus of hacker attention (e.g., *Runtime Process Infection* [37] for an in-depth intro to the `ptrace()` subsystem). Not surprisingly, hackers are the leading purveyors of specializes debuggers, such as *dumBug*, *Rasta Debugger*, and the *Immunity debugger* to name a few.

For Linux, a good example is *Handling Interrupt Descriptor Table for fun and profit* [38], which serves as both a concise introduction to the x86 interrupt system and its use on several composition-critical kernel paths, as well as its role in implementing various OS and debugging abstractions (including system calls and their place in the IDT). This approach was followed by a systematic study of particular interrupt handlers, such as the *Hijacking Linux Page Fault Handler* [39].

Overloading the page fault handler in particular has become a popular mechanism for enforcing policy in kernel hardening patches (e.g., *PaX*[4] and *OpenWall*[5]). However, other handlers have been overloaded as well, providing, e.g., support for enhanced debugging not relying on the kernel's standard facilities—and thus not conflicting with them and not registering with them, to counteract anti-debugging tricks. Since both rootkits (e.g., the proof-of-concept *DR Rootkit* that uses the x86 debug registers exclusively as its control flow mechanism) and anti-RE armored applications (e.g., Skype, cf. *Vanilla Skype* [40]; also, some commercial DRM products). In particular, the *Rasta Debugger* demonstrates such "unorthodox debugging" trap overloading-based techniques.

Notably, similar trap overloading techniques are used to expand the semantics of classic debugger breakpoint-able events. For instance, *OllyBone*[6] manipulated page translation to catch an instruction fetch from a page just written to, a typical behavior of a malware unpacker handing execution to the unpacked code. Note the temporal semantics of this composed trap, which was at the time beyond the capabilities of any debugger. A similar use of the ×86 facilities, and in particular the split instruction and data TLBs was used by the *Shadow Walker* [41] rootkit to cause code segments loaded by an antivirus analyzer to be fetched from a different physical page than the actual code, so that the analyzer could receive innocent data—a clever demonstration of the actual vs assumed nature of ×86 memory translation mechanism. For an in-depth exploration of just how powerful that mechanism can be, as well as for background on previous work, see Bangert [42].

4 Conclusion

Exploit engineers will show you the unintended limits of your system's functionality. If software engineers want to reduce this kind of latent functionality, they will have to begin understanding it as an artifact that supports the exploit engineer's workflow.

Software engineers should view their input data as "acting on code", not the other way around; indeed, in exploits inputs serves as a de-facto bytecode for execution environments that can be composed from the elements of their assumed runtime environment. Writing an exploit—creating such bytecode—is as structured a discipline as engineering "normal" software systems. As a process, it is no more arcane or unapproachable than the ways we currently use to write large software systems.

Yet, a significant challenge remains. If, as hinted above, we want to have a practical impact on the challenge of secure composition, can we actually train software engineers to see their input parameters and data formats *as bytecode* even as they specify it? Even as they bring it into existence, where it is by definition partially

[4] http://pax.grsecurity.net/

[5] http://www.openwall.com/Owl/

[6] http://www.joestewart.org/ollybone/

318981_1_en_13_chapter ☑ TYPESET ☐ DISK ☐ LE ☑ CP Disp.:24/3/2014 Pages: 15 Layout: T1-Standard

formulated, can they anticipate how it might be misused? We posit that this constant and frequent self-check is worth the effort: Software engineers should familiarize themselves with anti-security patterns lest preventable 'weird machines' arise in critical applications.

References

1. Bratus S, Locasto ME, Patterson ML, Sassaman L, Shubina A. Exploit programming: from buffer overflows to "weird machines" and theory of computation. login: Dec 2011.
2. Shacham H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the ×86). In: Proceedings of the 14th ACM conference on computer and communications security, CCS '07. New York: ACM; p. 552–561.
3. Roemer R, Buchanan E, Shacham H, Savage S. Return-oriented programming: systems, languages, and applications. ACM Trans Inf Syst Secur. 2012;15(1):2:1–2:34.
4. Dan R. Anatomy of a remote kernel exploit. http://www.cs.dartmouth.edu/-sergey/cs108/2012/Dan-Rosenberg-lecture.pdf (2011).
5. Holler C, Herzig K, Zeller A. Fuzzing with code fragments. In: Proceedings of the 21st USENIX conference on security symposium, Security'12. Berkeley: USENIX Association; 2012. p. 38–38.
6. Caballero Juan, Song Dawn. Automatic protocol reverse-engineering: message format extraction and field semantics inference. Comput Netw. 2013;57(2):451–74.
7. Samuel M, Erlingsson Ú. Let's parse to prevent pwnage invited position paper. In: Proceedings of the 5th USENIX conference on Large-scale exploits and emergent threats, LEET'12, Berkeley, USA: USENIX Association; 2012. p. 3–3.
8. Jana s, Shmatikov V. Abusing file processing in malware detectors for fun and profit. In: IEEE symposium on security and privacy'12; 2012. p. 80–94.
9. Xi W, Haogang C, Alvin C, Zhihao J, Nickolai Z, Kaashoek MF. Undefined behavior: what happened to my code? In: Proceedings of the Asia-Pacific workshop on systems, APSYS'12. New York, USA: ACM; 2012. p. 9:1–9:7.
10. Dullien T. Exploitation and state machines: programming the "weird machine", revisited. In: Infiltrate conference, Apr 2011.
11. Sassaman L, Patterson ML, Bratus S, Locasto ME, Shubina A. Security applications of formal language theory. Dartmouth College: Technical report; 2011.
12. Shapiro R, Bratus S, Smith SW. "Weird machines" in ELF: a Spotlight on the underappreciated metadata. In: 7th USENIX workshop of offensive technologies. https://www.usenix.org/system/files/conference/woot13/woot13-shapiro.pdf. 2013
13. Cesare. S. Shared library call redirection via ELF PLT, Infection. Dec 2000.
14. Sd, Devik. Linux On-the-fly Kernel patching without LKM, Dec 2001.
15. Mayhem. Understanding Linux ELF RTLD internals. http://s.eresi-project.org/inc/articles/elf-rtld.txt (2002).
16. Nergal. The advanced return-into-lib(c) Exploits: PaX Case Study. Phrack Mag. 2001;58(4).
17. Oakley J, Sergey B. Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code. In WOOT. 2011. p. 91–102.
18. Skape. Locreate: an anagram for relocate. Uninformed. 2007;6.
19. Sotirov A. Heap feng shui in javascript. In: Blackhat; 2007.
20. Redpantz. The art of exploitation: MS IIS 7.5 remote heap overflow. Phrack Mag. 68(12), Apr 2012.
21. Huku, Argp. The art of exploitation: exploiting VLC, a jemalloc case study. Phrack Maga. 2012;68(13).
22. Ferguson J. Advances in win32 aslr evasion, May 2011.

23. Bilar D. On callgraphs and generative mechanisms. J Comput Virol. 2007;3(4).
24. Richarte D. About exploits writing. Core security technologies presentation 2002.
25. Gera, Riq. Advances in format string exploitation. Phrack Mag. 2002;59(7).
26. One A. Smashing the stack for fun and profit. Phrack 1996;49:14. http://phrack.org/issues.html?issue=49&id=14.
27. Palmers. Sub proc_root auando sumus (Advances in Kernel hacking). Phrack 2001;58:6. http://phrack.org/issues.html?issue=58&id=6.
28. Palmers. 5 Short stories about execve (advances in Kernel hacking II). Phrack 2002;59:5. http://phrack.org/issues.html?issue=59&id=5.
29. Rutkowski JK. Execution path analysis: finding Kernel based rootkits. Phrack 2002;59:10. http://phrack.org/issues.html?issue=59&id=10.
30. Cesare S. Runtime Kernel kmem patching. 1998. http://althing.cs.dartmouth.edu/local/vsc07.html.
31. Mayhem. IA32 advanced function hooking. Phrack 2001;58:8. http://phrack.org/issues.html?issue=58&id=8.
32. Klog. Backdooring binary objects. Phrack 2000;56:9. http://phrack.org/issues.html?issue=56&id=9.
33. The Grugq. Cheating the ELF: subversive dynamic linking to libraries, 2000.
34. Mayhem. Understanding Linux ELF RTLD Internals, 2002. http://s.eresi-project.org/inc/articles/elf-rtld.txt.
35. Grugq, Scut. Armouring the ELF: binary encryption on the UNIX platform. Phrack ; 2001;58:5. http://phrack.org/issues.html?issue=58&id=5.
36. Rutkowska J. Passive covert channels implementation in Linux Kernel. 21st chaos communications congress, 2004. http://events.ccc.de/congress/2004/fahrplan/files/319-passive-covert-channels-slides.pdf.
37. (Anonymous author). Runtime process infection. Phrack 20025;9:8. http://phrack.org/issues.html?issue=59&id=8.
38. Kad. Handling interrupt descriptor table for fun and profit. Phrack 2002;59:4. http://phrack.org/issues.html?issue=59&id=4.
39. Buffer. Hijacking Linux page fault handler exception table. Phrack 2003;61:7. http://phrack.org/issues.html?issue=61&id=7.
40. Desclaux F, Kortchinsky K. Skype V. REcon. http://www.recon.cx/en/f/vskype-part1.pdf (2006).
41. Sparks S, Butler J. "Shadow Walker": Raising the bar for rootkit detection. BlackHat; 2005. http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf.
42. Bangert J, Bratus S, Rebecca S, Sean WS. The page-fault weird machine: lessons in instruction-less computation. In: 7th USENIX workshop of offensive technologies. Aug 2013. https://www.usenix.org/system/files/conference/woot13/woot13-bangert.pdf.

318981_1_en_13_chapter ☑ TYPESET ☐ DISK ☐ LE ☑ CP Disp.:24/3/2014 Pages: 15 Layout: T1-Standard