Composition Patterns of Hacking

Sergey Bratus*, Julian Bangert*, Alexandar Gabrovsky*, Anna Shubina*, Daniel Bilar†, and Michael E. Locasto‡
*Dartmouth College
‡Siege Technologies
‡The University of Calgary

***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 paper contains a top-level overview of these approaches and their historical development.

I. Introduction

When academic researchers study hacking, 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[1]). We argue that studying just these two classes is an insufficient means of making progress toward the goal of more fundamentally secure software because this perspective does not include a notion of *composing* the attacker computation with the native computation of the target. We claim that such composition is the source of the most powerful and productive concepts and methodologies that emerge from exploitation practice.

When researchers focus on attacker 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 paper we discuss not *hacking*, but *exploitation* or *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 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[2].

In this paper we review a series of classic exploitation[3] techniques from the perspective of composition (specifically, composition as the basic unit of activity of an engineering workflow, whether that workflow is a traditional software engineering workflow or a workflow focused on engineering an exploit). Many of these have been extensively described 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.

We posit that such analysis is required for designing defensible systems (see Section IV). 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.

II. A Tale of Two Engineering Workflows

Hacking, vulnerability analysis, and exploit programming are generally perceived to be difficult and arcane activities. The development of exploits is 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 paper asserts that what seems arcane is really only unfamiliar. In fact, although it may be difficult to conceive of

[1]Discussed in the hacker community since at least early 2000s, see [2] for a brief history sketch. These artifacts were brought to the attention of academia by Shacham [30], [24].

[2]It also serves as an excellent teaching aid in advanced OS courses; see, e.g., [25].

[3] Since our focus is on composition, we do not distinguish between techniques used by rootkits vs. exploits. Indeed, rootkits, even when introduced at higher privilege, face similar context limitation obstacles as exploits.

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 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 Section III.

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.

A. 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 toolchains, 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.

B. 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 (i.e., weird machine programming) 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. 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.

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).

III. Patterns

a) Exploitation as programming "weird machines": Bratus et al. [2] 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 bugs, features 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 [7], from a theory standpoint in Sassaman [28].

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 thread of control is simultaneously used by others, and involves dissecting and "slimming down" interfaces to their actual implementation primitives and finding out unintended yet stable properties of these primitives.

b) Recoving 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[4]. 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.

Cesare [5] 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., [29]. In [16] mayhem builds on the same idea by looking into the significance and priority of ELF's `.dynamic` symbols. Nergal [17] co-opted Linux's own dynamic linker into an ROP[5] crafted stack frame-chaining scheme, to have necessary symbols resolved and libraries loaded. Oakley [18] showed how to co-opt the DWARF-based exception handling mechanism.

Skape [31] takes the understanding of ELF in a different direction by showing how its relocation mechanism works and how that could be used for unpacking a binary.

c) 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 [32] 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 [22], [11] 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., [8]).

d) Seeing through 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 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 [17], who modifies the return addresses on the stack to compose existing code elements into an exploit, and the LOCREATE [31] packer which obfuscates binary code by using the primitives

[4]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.

[5]Which it pre-dates, together with other hacker descriptions of the technique, by 5-7 years.

for dynamic linking.

e) Scoping Out The Devil's Bargain:

> 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 bargain with the devil. 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 [23], [9], [7]. We posit that understanding this gap is the way to more secure API design.

f) 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 [19] 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)* [21] (VFS structures and their linking and hijacking), *5 Short Stories about execve (Advances in Kernel Hacking II)* [20] (driver/DDK interfaces, different binary format support), and *Execution path analysis: finding kernel based rootkits* [27] (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* [4] 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* [14]).

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 [13]*), the control flow of dynamic linking (e.g., *Cheating the ELF* [34] and *Understanding Linux ELF RTLD internals* [15]), 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* [10]).

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* [26] 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.

g) "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* [1] 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* [12], 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* [3].

Overloading the page fault handler in particular has become a popular mechanism for enforcing policy in kernel hardening patches (e.g., *PaX* and *OpenWall*); 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* [6]; 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 x86 facilities, and in particular the split instruction and data TLBs was used by the *Shadow Walker* [33] 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 x86 memory translation mechanism.

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

IV. 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 formulated, can they anticipate how it might be misused? Is this constant and frequent self-check worth the effort, or should software engineers first build a system without regard to analyzing anti-security composition patterns?

Acknowledgment

This paper would not be possible without the efforts of Felix Lindner, Hugo Fortier, Bruce Potter, David Hulton, Enno Rey and other organizers of conferences where many of the results we discussed have been presented, and many discussions took place.

References

[1] (anonymous author). Runtime Process Infection. *Phrack* 59:8. http://phrack.org/issues.html?issue=59&id=8.

[2] Sergey Bratus, Michael E. Locasto, Meredith L. Patterson, Len Sassaman, and Anna Shubina. Exploit programming: from buffer overflows to "weird machines" and theory of computation. *;login:*, December 2011.

[3] buffer. Hijacking Linux Page Fault Handler Exception Table. *Phrack* 61:7. http://phrack.org/issues.html?issue=61&id=7.

[4] Silvio Cesare. Runtime Kernel kmem Patching. http://althing.cs.dartmouth.edu/local/vsc07.html.

[5] Silvio Cesare. Shared Library Call Redirection via ELF PLT Infection, Dec 2000.

[6] Fabrice Desclaux and Kostya Kortchinsky. Vanilla Skype. REcon 2006. http://www.recon.cx/en/f/vskype-part1.pd, http://www.recon.cx/en/f/vskype-part2.pdf.

[7] Thomas Dullien. Exploitation and state machines: Programming the "weird machine", revisited. In *Infiltrate Conference*, Apr 2011.

[8] Justin Ferguson. Advances in win32 aslr evasion, May 2011.

[9] gera and riq. Advances in Format String Exploitation. *Phrack Magazine*, 59(7), Jul 2002.

[10] grugq and scut. Armouring the ELF: Binary encryption on the UNIX platform. *Phrack* 58:5. http://phrack.org/issues.html?issue=58&id=5.

[11] huku and argp. The Art of Exploitation: Exploiting VLC, a jemalloc Case Study. *Phrack Magazine*, 68(13), Apr 2012.

[12] kad. Handling Interrupt Descriptor Table for fun and profit. *Phrack* 59:4. http://phrack.org/issues.html?issue=59&id=4.

[13] klog. Backdooring Binary Objects. *Phrack* 56:9. http://phrack.org/issues.html?issue=56&id=9.

[14] mayhem. IA32 Advanced Function Hooking. *Phrack* 58:8. http://phrack.org/issues.html?issue=58&id=8.

[15] mayhem. Understanding Linux ELF RTLD Internals. http://s.eresi-project.org/inc/articles/elf-rtld.txt.

[16] mayhem. Understanding Linux ELF RTLD internals. http://s.eresi-project.org/inc/articles/elf-rtld.txt, Dec 2002.

[17] Nergal. The Advanced return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine*, 58(4), Dec 2001.

[18] James Oakley and Sergey Bratus. Exploiting the hard-working dwarf: Trojan and exploit techniques with no native executable code. In *WOOT*, pages 91–102, 2011.

[19] Aleph One. Smashing the Stack for Fun and Profit. *Phrack* 49:14. http://phrack.org/issues.html?issue=49&id=14.

[20] palmers. 5 Short Stories about execve (Advances in Kernel Hacking II). *Phrack* 59:5. http://phrack.org/issues.html?issue=59&id=5.

[21] palmers. Sub proc_root Quando Sumus (Advances in Kernel Hacking). *Phrack* 58:6. http://phrack.org/issues.html?issue=58&id=6.

[22] redpantz. The Art of Exploitation: MS IIS 7.5 Remote Heap Overflow. *Phrack Magazine*, 68(12), Apr 2012.

[23] Gerardo Richarte. About Exploits Writing. Core Security Technologies Presentation, 2002.

[24] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications, 2009.

[25] Dan Rosenberg. Anatomy of a remote kernel exploit. http://www.cs.dartmouth.edu/~sergey/cs108/Dan-Rosenberg-lecture.pdf.

[26] Jonna Rutkowska. 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.

[27] Jan K. Rutkowski. Execution Path Analysis: Finding Kernel Based Rootkits. *Phrack* 59:10. http://phrack.org/issues.html?issue=59&id=10.

[28] Len Sassaman, Meredith L. Patterson, Sergey Bratus, Michael E. Locasto, and Anna Shubina. Security applications of formal language theory. Technical report, Dartmouth College, 2011.

[29] sd and devik. Linux On-the-fly Kernel Patching without LKM, Dec 2001.

[30] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: return-into-libc without Function Calls. In *ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[31] skape. Locreate: an Anagram for Relocate. *Uninformed*, 6, Jan 2007.

[32] Alexander Sotirov. Heap feng shui in javascript. In *Blackhat 2007*, 2007.

[33] Sherri Sparks and Jamie Butler. "Shadow Walker": Raising The Bar For Rootkit Detection. BlackHat 2005. http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf.

[34] the grugq. Cheating the ELF: Subversive Dynamic Linking to Libraries. althing.cs.dartmouth.edu/local/subversiveld.pdf.