G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries

Kaan Onarlioglu
Bilkent University, Ankara
onarliog@cs.bilkent.edu.tr

Leyla Bilge
Eurecom, Sophia Antipolis
bilge@eurecom.fr

Andrea Lanzi
Eurecom, Sophia Antipolis
lanzi@eurecom.fr

Davide Balzarotti
Eurecom, Sophia Antipolis
balzarotti@eurecom.fr

Engin Kirda
Eurecom, Sophia Antipolis
kirda@eurecom.fr

ABSTRACT

Despite the numerous prevention and protection mechanisms that have been introduced into modern operating systems, the exploitation of memory corruption vulnerabilities still represents a serious threat to the security of software systems and networks. A recent exploitation technique, called Return-Oriented Programming (ROP), has lately attracted a considerable attention from academia. Past research on the topic has mostly focused on refining the original attack technique, or on proposing partial solutions that target only particular variants of the attack.

In this paper, we present G-Free, a compiler-based approach that represents the first practical solution against any possible form of ROP. Our solution is able to eliminate all unaligned free-branch instructions inside a binary executable, and to protect the aligned free-branch instructions to prevent them from being misused by an attacker. We developed a prototype based on our approach, and evaluated it by compiling `GNU libc` and a number of real-world applications. The results of the experiments show that our solution is able to prevent any form of return-oriented programming.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection

General Terms

Security

Keywords

Return-oriented programming, ROP, return-to-libc

1. INTRODUCTION

As the popularity of the Internet increases, so does the number of attacks against vulnerable services [3]. A common way to compromise an application is by exploiting memory corruption vulnerabilities to transfer the program execution to a location under the control of the attacker. In these kinds of attacks, the first step requires to find a technique to overwrite a pointer in memory. Overflowing a buffer on the stack [5] or exploiting a format string vulnerability [26] are well-known examples of such techniques. Once the attacker is able to hijack the control flow of the application, the next step is to take control of the program execution to perform some malicious activity. This is typically done by injecting in the process memory a small payload that contains the machine code to perform the desired task.

A wide range of solutions have been proposed to defend against memory corruption attacks, and to increase the complexity of performing these two attack steps [10, 11, 12, 18, 35]. In particular, all modern operating systems support some form of memory protection mechanism to prevent programs from executing code that resides in certain memory regions [33]. The goal of this technique is to protect against code injection attacks by setting the permissions of the memory pages that contain data (such as the stack and the heap of the process) as non-executable.

One of the techniques to bypass non-executable memory without relying on injected code involves reusing the functionality provided by the exploited application. Using this technique, which was originally called return-to-lib(c) [31], an attacker can prepare a fake frame on the stack and then transfer the program execution to the beginning of a library function. Since some popular libraries (such as the `libc`) contain a wide range of functionality, this technique is sufficient to take control of the program (e.g., by exploiting the `system` function to execute `/bin/sh`).

In 2007, Shacham [29] introduced an evolution of return-to-lib(c) techniques [23, 27, 31] called Return-Oriented Programming (ROP). The main contribution of ROP is to show that it is possible for an attacker to execute arbitrary algorithms and achieve Turing completeness without injecting any new code inside the application.

The idea behind ROP is simple: Instead of jumping to the beginning of a library function, the attacker chains together existing sequences of instructions (called Gadgets) that have been previously identified inside existing code. The large availability of gadgets in common libraries allows the attacker to implement the same functionality in many different ways. Thus, removing potentially dangerous functions (e.g., `system`) from common libraries is ineffective against ROP, and does not provide any additional security.

ROP is particularly appealing for rootkit development since it can defeat traditional defense techniques based on kernel data integrity [36] or code verification [24, 28]. Another interesting domain is related to exploiting architectures with immutable memory protection (e.g., to compromise electronic voting machines as shown in [7]). ROP was also recently adopted by real attacks observed in the wild as a way to bypass Windows' Data Execution Prevention (DEP) technology [2].

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

The great interest around ROP quickly evolved into an arms race between researchers. On the one side, the basic attack technique was extended to various processor architectures [6, 7, 14, 15, 34] and the feasibility of mounting this attack at the kernel level was demonstrated [19]. On the other side, ad-hoc detection and protection mechanisms to mitigate the attack were proposed [9, 13, 16, 22]. To date, existing solutions have focused only on the basic attack, by detecting, for instance, the anomalous frequency of return instructions executed [9, 16], or by removing the `ret` opcode to prevent the gadget creation [21]. Unfortunately, a recent advancement in ROP [8] has already raised the bar by adopting different instructions to chain the gadgets together, thus making all existing protection techniques ineffective.

In this paper, we generalize from all the details that are specific to a particular exploitation technique to undermine the foundation on top of which return-oriented programming is built: the availability of instruction sequences that can be reused by an attacker. We present a general approach for the IA-32 instruction set that combines different techniques to eliminate all possible sources of reusable instructions. More precisely, we use code rewriting techniques to remove all unaligned instructions that can be used to link the gadgets. Moreover, we introduce a novel protection technique to prevent the attacker from misusing existing return or indirect jump/call instructions.

We implemented our solution under Linux as a pre-processor for the popular `GNU Assembler`. We then evaluated our tool on different real-world applications, with a special focus on the `GNU libc` (`glibc`) library. Our experiments show that our solution can be applied to complex programs, and it is able to remove all possible gadgets independently from the mechanism used to connect them together. A program compiled with our system is, on average, 26% larger and 3% slower (when all the linked libraries are also compiled with our solution). This is a reasonable overhead that is in line with existing stack protection mechanisms such as StackGuard [11].

This paper makes the following contributions:

- We present a novel approach to prevent an attacker from reusing fragments of existing code as basic blocks to compose malicious functionality.
- To the best of our knowledge, we are the first to propose a general solution to defeat all forms of ROP. That is, our solution can defend against both known variations and future evolutions of the attack.
- We developed *G-Free*, a proof-of-concept implementation to generate programs that are hardened against return-oriented programming. Our solution requires no modification to the application source code, and can also be applied to system applications that contain large sections of assembly code.
- We evaluated our technique by compiling gadget-free versions of `glibc` and other real-world applications.

The rest of the paper is structured as follows: In Section 2, we analyze the key concepts of return-oriented programming.In Section 3, we summarize proposed defense techniques against memory corruption attacks and ROP. In Section 4, we present our approach for compiling gadget-free applications. In Section 5, we describe our prototype implementation. In Section 6, we show the results of the experiments we conducted for evaluating the impact and performance of our system. Finally, in Section 7, we briefly conclude the paper.

2. GADGETS

Before presenting the details of our approach, we establish a more precise and general model for the class of attacks we wish to prevent. Therefore, we generalize the concept of return-oriented programming by abstracting away from all the details that are specific to a particular attack technique.

2.1 Programming with Gadgets

The core idea of return-oriented programming is to "borrow" sequences of instructions from existing code (either inside the application or in the linked libraries) and chaining them together in an order chosen by the attacker. Therefore, in order to use this technique, the attacker has to first identify a collection of useful instruction sequences that she can later reuse as basic blocks to compose the code to be executed. A crucial factor that differentiates return-oriented programming from simpler forms of code reuse (such as traditional return-to-lib(c) attacks) is that the collection of code snippets must provide a comprehensive set of functionalities that allows the attacker to achieve Turing completeness *without* injecting any code [29]. The second step of ROP involves devising a mechanism to manipulate the control flow in order to chain these code snippets together, and build meaningful algorithms.

Note that these two requirements are not independent: To allow the manipulation of the control flow, the instruction sequences must exhibit certain characteristics that impose constraints on the way they are chosen. For example, sequences may have to terminate with a return instruction, or they may have to preserve the content of a certain CPU register. In this paper, we use the term *Gadget* to refer to any valid sequence of instructions that satisfies the control flow requirements.

In a traditional ROP attack, the desired control flow is achieved by placing the addresses of the gadgets on the stack and then exploiting `ret` instructions to fetch and copy them to the instruction pointer. In other words, if we consider each gadget as a monolithic instruction, the stack pointer plays the role of the instruction pointer in a normal program, transferring the control flow from one gadget to the next. Consequently, gadgets are initially defined by Shacham as useful snippets of code that terminate with a `ret` instruction [29].

However, the use of `ret` instructions is just one possible way of chaining gadgets together. In a recent refinement of the technique [8], Checkoway and Shacham propose a variant of ROP in which *return-like* instructions are employed to fetch the addresses from the stack. Because these sequences are quite rare in regular binaries, indirect jumps (e.g., `jmp *%eax`) are used as gadget terminators to jump to a previously identified return-like sequence. In theory, it is even possible to design control flow manipulation techniques that are not stack-based, but that store values in other memory areas accessible at runtime by an attacker (e.g., on the heap or in global variables).

As a result, in order to find a general solution to the ROP threat, we need to identify a property that all possible variants of return-oriented programming have in common. Kornau [34] identified such a property in the fact that every gadget, in order to be reusable, has to end with a "*free-branch*" instruction, i.e., an instruction that can change the program control flow to a destination that is (or that can be under certain circumstances) controlled by the attacker. According to this definition, in each gadget, we can recognize two parts: the *code section* that implements the gadget's functionality and the *linking section* that contains the instructions used to transfer the control to the next gadget. The linking section needs to end with a free branch, but it can also contain additional instructions. For instance, a possible linking section could be the following se-

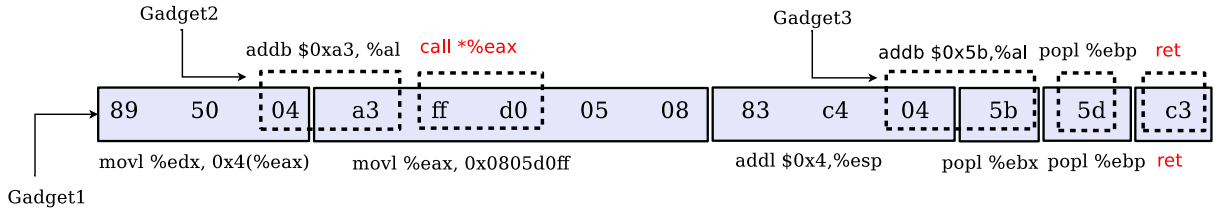

Figure 1: Examples of different gadgets that can be extracted from a real byte sequence

quence: `pop %ebx; call *%ebx`.

2.2 Gadget Construction

In the x86 architecture, gadgets are not limited to sequences of existing instructions. In fact, since the IA-32 instruction set does not have fixed length instructions, the opcode that will be executed depends on the starting point of the execution in memory. Therefore, the attacker can build different gadgets by jumping inside existing instructions.

Figure 1 shows how, depending on the alignment of the first and last instruction, it is possible to construct three different kinds of gadgets. `Gadget1` is an *aligned* gadget that only uses "intended" instructions already present in the function code. `Gadget2` is a gadget that contains only "unaligned" instructions ending with the unintended `call *%eax`. Finally, `Gadget3` starts by using an unintended `add` instruction, then re-synchronizes with the normal execution flow, and ends by reaching the function return. This example demonstrates how a short sequence of 14 bytes can be used for constructing many possible gadgets. Considering that a common library such as `libc` contains almost 18K free branch instructions and that each of them can be used to construct multiple gadgets, it is not difficult for an attacker to find the functionality he needs to execute arbitrary code.

If we can prevent the attacker from finding useful instruction sequences that terminate with a free branch, we can prevent any return-oriented programming technique. We present our approach to reach this goal in Section 4.

3. RELATED WORK

Several defense mechanisms attempt to detect memory exploits which represent a fundamental basic block for mounting return-to-lib(c) attacks. StackGuard [11] and ProPolice [18] are compile-time solutions that aim at detecting stack overflows. PointGuard encrypts pointers stored in memory to prevent them from being corrupted [10]. StackShield [35] and StackGhost [17] use a shadow return address stack to save the return addresses and to check whether they have been tampered with at function exits. A complete survey of traditional mitigation techniques together with their drawbacks is presented in [12]. Our solution, in order to avert ROP attacks, prevents tampering with the return address as well; but it does not target other memory corruption attacks.

One of the most effective techniques that hamper return-to-lib(c) attacks is Address Space Layout Randomization (ASLR) [32]. In its general form, this technique randomizes positions of stack, heap, and code segments together with the base addresses of dynamic libraries inside the address space of a process. Consequently, an attacker is forced to correctly guess the positions where these data structures are located to be able to mount a successful attack. Despite the better protection offered by this mechanism, researchers showed that the limited entropy provided by known ASLR implementations can be evaded either by performing a brute-force attack on 32-bit architectures [30] or by exploiting Global Address Table and de-randomizing the addresses of target functions [25].

Various approaches proposed by the research community aim at impeding ROP attacks by ensuring the integrity of saved return addresses. Frantsen et al. [17] presented a shadow return address stack implemented in hardware for the Atmel AVR microcontroller, which can only be manipulated by `ret` and `call` instructions. ROPdefender [22] uses runtime binary instrumentation to implement a shadow return address stack where saved return addresses are duplicated and later compared with the value in the original stack at function exits. Even though ROPdefender is suitable for impeding basic ROP attacks, it suffers from performance issues due to the fact that the system checks every machine instruction executed by a process.

Another method, called program shepherding [20], can prevent basic forms of ROP as well as code injection by monitoring control flow transfers and ensuring library code is entered from exported interfaces.

Other approaches [9, 13] aim to detect ROP-based attacks relying on the observation that running gadgets results in execution of short instruction sequences that end with frequent `ret` instructions. They proposed to use dynamic binary instrumentation to count the number of instructions executed between two `ret` opcodes. An alert is raised if there are at least three consecutive sequences of five or fewer instructions ending with a `ret`.

The most similar approach to ours is a compiler-based solution developed in parallel to our work by Li et al. [21]. This system eliminates unintended `ret` instructions through code transformations, and instruments all `call` and `ret` instructions to implement return address indirection. Specifically, each call instruction is modified to push onto the stack an index value that points to a return address table entry, instead of the return address itself. Then, when a `ret` instruction is executed, the saved index is used for looking up the return address from the table. Although this system is more efficient compared to the previous defenses, it is presented as a solution specifically tailored for gadgetless kernel compilation, and it exploits characteristics of kernel code for gadget elimination and increased performance. Moreover, the implementation requires manual modifications to all the assembly routines.

It is important to note that none of the defenses proposed so far can address more advanced ROP attacks that utilize free-branch instructions different from `ret`. The solution we present in this paper is the first to address all free-branch instructions, and the first that can be applied at compile-time to protect any program from ROP attacks.

4. CODE WITHOUT GADGETS

Our goal is to provide a proactive solution to build gadget-free executables that cannot be targeted by any possible ROP attack. In particular, we strive to achieve a *comprehensive*, *transparent*, and *safe* solution. By *comprehensive*, we mean that we would like our solution to eliminate all possible gadgets by removing the linking

mechanisms that are necessary to chain instruction sequences together. *Transparent* means that this process must require no intervention from the user, such as manual modifications to the source code. Finally, we would like to present a solution that is *safe*: That is, it should preserve the semantics of the program, be compatible with compiler optimizations, and support applications that contain routines written in assembly language.

In order to reach our goals, we devise a compiler-based approach that first eliminates all unaligned free-branch instructions inside a binary executable, and then protects the aligned free-branch instructions to prevent them from being misused by an attacker.

We achieve the first point through a set of code transformation techniques that ensure free-branch instructions never appear inside any legitimate aligned instruction. This leaves the attacker with the only option of exploiting existing `ret` and `jmp*/call*` instructions. To eliminate this possibility, we introduce a mechanism that protects these potentially dangerous instructions by ensuring that they can be executed only if the functions in which they reside were executed from their entry points.

Consequently, an attacker can only execute entire functions from the start to the end as opposed to running arbitrary code. This effectively de-generalizes the threat to a traditional return-to-lib(c) attack, eliminating the advantages of achieving Turing completeness without injecting any code in the target process.

Our approach uses a combination of techniques, namely alignment sleds, return address encryption, frame cookies and code rewriting. The rest of this section describes each technique in detail.

4.1 Free Branch Protection

The first set of techniques aim to protect the aligned free-branch instructions available in the binary. These include the actual `ret` instructions at the end of each function and the `jmp*/call*` instructions that are sometimes present in the code.

Unfortunately, these instructions cannot be easily eliminated without altering the application's behavior. In addition, replacing them with semantically equivalent pieces of code is likely not going to solve the problem because the attacker could still use the replacements to achieve the same functionality.

Therefore, we propose a simple solution inspired by existing stack protection mechanisms (e.g., StackGuard [11]). The goal is to instrument functions with short blocks of code to ensure that aligned free-branch instructions can only be executed if the running function has been entered from its proper entry point. In particular, we employ two complementary techniques: an efficient return address encryption to protect `ret` instructions, and a more sophisticated cookie-based technique we additionally apply only to those functions that contain `jmp*/call*` instructions. In Section 4.3, we discuss the possibility that an attacker attempts to exploit these protection blocks, and in Section 5.5 we show how we avoid this threat in our prototype.

Finally, we prepend the code performing the checks with *alignment sleds*. Alignment sleds are special sequences of bytes by which we enforce aligned execution of a set of critical instructions. In particular, we use this technique to prevent an attacker from bypassing our free branch protection code by executing it in an unaligned fashion.

4.1.1 Alignment Sleds

An alignment sled is a sufficiently-long sequence of bytes, encoding one or more instructions that have no effect on the status of the execution. Its length is set to ensure that regardless of the alignment prior to reaching the sled, the execution will eventually land on the sled and execute it until the end. Even if an attacker jumps into the binary at an arbitrary point and executes a number of unaligned instructions, when she reaches the sled, the execution will be forced to realign with the actual code. Thus, it will never reach any unintended opcode present in the instructions following the sled.

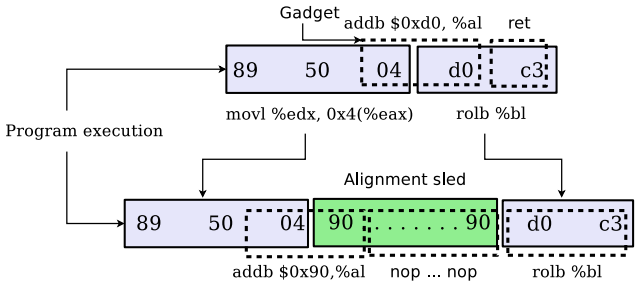

Figure 2: Application of an alignment sled to prevent executing an unaligned `ret` (`0xc3`) instruction

The simplest way to implement an alignment sled is to use a sequence of `nop` instructions (see Figure 2 for an example). The number of `nop` instructions must be determined by taking into consideration the maximum number of consecutive `nop` bytes (`0x90`) that can tail a valid instruction. If we set the length to anything less than that, an attacker could find an unintended instruction that encompasses the whole sled and any number of bytes from the following instruction, in which case the execution will continue in an unaligned fashion. In the IA-32 architecture, the longest such sequence becomes possible when we have both an address displacement and an immediate value entirely composed of `0x90` bytes [4], which makes a total of 8 bytes. Additionally, we can have either a ModR/M byte, a SIB byte or an opcode with the value `0x90` (but only one of them at a time). As a result, we can safely set the number of `nop` instructions in our sled to 9.

Note that the sled length calculation presented in this section is an over-approximation: By also taking into account the bytes preceding the sled and which instructions they can possibly encode, it is possible to automatically compute the required sled length case-by-case.

Finally, we prepend the sled with a relative jump instruction to skip over the sled bytes. Consequently, if the execution is already aligned it will hit the jump and not incur the performance penalty of executing the sequence of `nop` instructions.

4.1.2 Return Address Protection

This technique involves instrumenting entry points of the functions that contain `ret` instructions with a short header that encrypts the saved return address stored on the stack. Before `ret` instructions, we then insert a corresponding footer to restore the return address to its original value. If an attacker jumps into a function at an arbitrary position and eventually reaches our footer, the decryption routine processes the unencrypted return address provided by the attacker, computes an invalid value and the following `ret` instruction attempts to transfer the execution flow to an incorrect address that the attacker cannot control. This technique is similar to the random XOR canary implemented by StackGuard [11].

The encryption method we utilize is a simple exclusive-or of the return address with a *random key*. Since this solution does not affect the layout of the stack in any way, it does not require any further modifications to the function code.

4.1.3 Frame Cookies

In order to prevent the attacker from using existing `jmp*/call*` instructions, we need to adopt another protection mechanism. To

ModR/M	Operand 1	Operand 2
0xc2	%eax, %ax, %al	%edx, %dx, %dl
0xc3	%eax, %ax, %al	%ebx, %bx, %bl
0xca	%ecx, %cx, %cl	%edx, %dx, %dl
0xcb	%ecx, %cx, %cl	%ebx, %bx, %bl

SIB	Base	Scaled Index
0xc2	%edx	%eax*8
0xc3	%ebx	%eax*8
0xca	%edx	%ecx*8
0xcb	%ebx	%ecx*8

Table 1: ModR/M and SIB values encoding `ret` opcodes

this end, we instrument entry points of the functions that contain `jmp*/call*` instructions with an additional header to compute and push a random cookie onto the stack. This cookie is an exclusive-or of a *random key* generated at runtime and a *per-function constant* generated at compile time. The constant is used for uniquely identifying the function and it does not need to be kept secret.

Then, we prepend all the `jmp*/call*` instructions with a *validation block* which fetches the cookie, decrypts it, and compares the result with the *per-function constant*. If the cookie is not found or the values do not match, we invalidate the jump/call destination causing the application to crash. Finally, in the function footer, we insert a simple instruction to remove the cookie from the stack.

A significant consequence of this technique is that it alters the layout of the stack by storing an additional value. This requires us to fix the memory offsets of some of the instructions that access the stack according to the location where we store the cookie (we discuss the details of this issue in Section 5).

4.2 Code Rewriting

The second set of techniques we adopt in our approach focus on removing any unaligned free-branch instructions.

In the IA-32 architecture, instructions consist of some or all of the following fields: instruction prefixes, an opcode, a ModR/M byte, a SIB (Scale-Index-Base) byte, an address displacement, and finally, an immediate value. A `ret` instruction can be encoded with any of the `0xc2`, `0xc3`, `0xca` or `0xcb` bytes, and, as such, can be part of any of the instruction fields (excluding the prefixes). On the other hand, `jmp*/call*` instructions are encoded by two-byte opcodes: an `0xff` followed by an ModR/M byte carrying certain three-bit sequences. Hence, in addition to appearing inside a single instruction, they can also be obtained by a combination of two bytes coming from two consecutive instructions.

In this section, we discuss the various cases and describe the code rewriting techniques we use to eliminate all unintended free-branch opcodes.

4.2.1 Register Reallocation

The ModR/M and the SIB bytes are used for encoding the addressing mode and operands of an instruction. The use of certain registers as operands cause either the ModR/M or the SIB byte to be set to a value that corresponds to a `ret` opcode. The possible undesired encodings of these bytes are shown in Table 1. For instance, an instruction that specifies `%eax` as the source operand and `%ebx` as the destination, such as `movl %eax, %ebx`, assigns the value `0xc3` to the ModR/M byte. Similarly, using `%edx` as the base and `(%ecx * 8)` as the scaled index, the instruction `addl $0x2a,(%edx,%ecx,8)` contains `0xca` in its SIB byte.

In order to eliminate the unintended `ret` opcodes that result from such circumstances, we must avoid all of the undesired register pairings listed in Table 1. We achieve this by manipulating the register allocation performed during compilation to ensure that those pairs of registers never appear together in a generated instruction. When we detect such an instruction, we can perform the compiler's register allocation stage again, this time enforcing a different register assignment. As an alternative, we can perform a local reallocation by temporarily swapping the contents of the original operand with a new register, and then rewriting the instruction with this new register as its operand. In this way, we can bring forth an acceptable register pairing for the same instruction.

Finally, in some cases, the ModR/M byte could be used to specify an opcode extension and a single register operand. Sometimes, it is possible to rewrite these instructions using the same techniques described above to replace the register operand with a different one. However, floating point instructions can use implicit operands that cannot be substituted with others (e.g, `fld %st(2)`). Since all these instructions can have the `ret` opcode only in their second byte, we instead prepend them with an alignment sled. This leaves to the attacker only one byte (the opcode that specifies the FPU instruction) before the unaligned `ret`, and it is therefore impossible to use this byte to create any gadget.

4.2.2 Instruction Transformations

`ret` bytes appear in opcodes encoding `movnti (0x0f 0xc3)` and `bswap (0x0f 0xc8+<register_identifier>)` instructions. In the first case, `movnti` acts like a regular `mov` operation except that it uses a *non-temporal hint* to reduce cache pollution. Thus, we can safely replace it with a regular *mov* without any significant consequence. For the second, the opcode is determined according to the operand register and can encode a `ret` byte when certain registers are specified as the operand; therefore, as described in the previous section, we can perform a register reallocation to choose a different operand and obtain a safe `bswap` opcode.

4.2.3 Jump Offset Adjustments

Jump and call instructions may contain free-branch opcodes when using immediate values to specify their destinations. For instance, `jmp .+0xc8` is encoded as "`0xe9 0xc3 0x00 0x00 0x00`".

A free-branch opcode can appear at any of the four bytes constituting the jump/call target. If the opcode is the least significant byte, it is sufficient to append the forward jump/call with a single `nop` instruction (or prepend it if it is a backwards jump/call) in order to adjust the relative distance between the instruction and its destination:

```
jmp .+0xc8   ⇒   jmp .+0xc9
                 nop
```

However, when the opcode is at a different byte position, the number of `nop` instructions we need to insert increase drastically (256 for the second, 64K for the third and 16M for the last byte).

Fortunately, it is highly uncommon to have a free-branch opcode in one of the most significant bytes. For example, a jump offset encoded by "`0x00 0x00 0xc3 0x00`" indicates a 12MB forward jump. Considering the fact that jump instructions are ordinarily used for local control flow transitions inside a function, a 12MB offset would be infeasible in practice. Even if we were to come across such an offset, it is still possible to relocate the functions or code chunks addressed by the instruction to remove the opcodes.

4.2.4 Immediate and Displacement Reconstructions

Several arithmetic, logic and comparison operations can take immediate values as an operand, which may contain free-branch instruction opcodes. We can remove these by substituting the instruction with a sequence of different instructions that construct the immediate value in steps while carrying the same semantics. The fol-

lowing examples demonstrate the reconstruction process, assuming that %ebx is free or has been saved beforehand:

```
addl $0xc2, %eax   ⇒   addl $0xc1, %eax
                       inc %eax

xorb $0xca, %al    ⇒   movb $0xc9, %bl
                       incb %bl
                       xorb %bl, %al
```

Instructions that perform memory accesses can also contain free-branch instruction opcodes in the displacement values they specify (e.g., `movb %al, -0x36(%ebp)` represented as "0x88 0x45 0xca"). In such cases, we need to substitute the instruction with a semantically equivalent instruction sequence that uses an adjusted displacement value to avoid the undesired bytes. We achieve this by setting the displacement to a safe value and then compensating for our changes by temporarily adjusting the value in the base register. For example, we can perform a reconstruction such as:

```
                                   incl %ebp
movb $0xal, -0x36(%ebp)   ⇒        movb %al, -0x37(%ebp)
                                   decl %ebp
```

4.2.5 *Inter-Instruction Barriers*

Unintended `jmp*/call*` opcodes can result from the combination of two consecutive instructions. This happens when the last byte of an instruction is `0xff` and the first byte of the following instruction encodes a suitable opcode extension. We can remove these unintended `jmp*/call*` opcodes by inserting a *barrier* between two such instructions, effectively separating them and destroying the unintended opcode. For the barrier, the trivial choice of a `nop` instruction is not suitable since an `0xff` followed by a `0x90` still encodes an indirect call. Thus, we have to choose a safe nop-like alternative, such as "`movl %eax, %eax`".

4.3 Limitations of the Approach

By applying the techniques presented in this section, it is possible to remove all unaligned free-branch instructions from a binary, and to protect the aligned ones from being misused by an attacker. However, since our protection mechanism does not remove the free branches, but prepends a short piece of code to protect them, the result of the compilation will still contain some gadgets.

In fact, an attacker may skip the alignment sled by directly jumping *into* the return address or indirect jump/call protection blocks. This may result in executing a useful instruction sequence (intended or unintended) which terminates at the free-branch instruction we intend to protect.

However, since our approach only requires inserting two very short pieces of code, the number of possible gadgets that can be built is very limited and the gadget sizes are restricted to few instructions. By keeping this issue in mind, it is, therefore, possible to specifically craft the return address and indirect jump/call protection blocks to make sure they do not contain any convenient gadgets.

In particular, we discuss the techniques we used in our prototype implementation and the number and type of gadgets that are left in the applications compiled by our tool in Section 5.5.

5. IMPLEMENTATION

Our implementation efforts primarily focus on creating a fully-automated system that would not require any modifications to the program's source code or to the existing compilation tools. Unfortunately, system-wide libraries, which are the primary targets of ROP attacks, often rely on hand-tuned assembly routines to perform low-level tasks. This makes a pure compiler-based solution unable to intercept part of the final code. Therefore, we implemented our prototype in two separate components: an assembly code pre-processor designed to work as a wrapper for the `GNU Assembler (gas)`, and a simple binary analyzer responsible for gathering some information that is not available in the assembly source code.

In this section, we describe G-Free, a prototype system we developed based on the techniques presented in Section 4, and we discuss some of the issues we encountered while compiling `glibc` using our prototype.

5.1 Assembly Code Pre-Processor & Binary Analyzer

The assembly code pre-processor intercepts the assembly code generated by `cc1` (the `GNU C compiler` included in the `GNU Compiler Collection`) or coming directly from an assembly language source file. It then performs the required modifications to remove all the possible gadgets, and finally passes the control to the actual `gas` assembler. We must stress that in this implementation we modify neither the compiler nor the assembler; both are completely oblivious to the existence of our pre-processing stage. We only replace the `gas` executable with a small wrapper responsible for invoking our pre-processor before executing the assembler.

Our system successfully handles assembly routines written using non-standard programming practices. It supports position independent code (PIC) and compiler optimizations, including all of the `GCC` standard optimization levels (in fact, `glibc` *does not* compile if `GCC` optimizations are disabled).

There is one significant implication of directly working with assembly code: Our pre-processor is not exposed to the numeric values of immediate operands and memory displacements since these are often represented by symbolic values until linkage. Thus, it is not possible for us to identify all of the instructions that contain unintended free-branch opcodes just by looking at the assembly code. In order to address this issue, we use a two-step compilation approach. First, our system compiles a given program without doing any modifications to the original code. During this compilation, our pre-processor tags each of the instructions that contain immediate values or displacements with unique symbols. This information is then exported in the final executable's symbol table. In a second step, we use a binary analyzer to read the symbol table of the executable and check whether any of the instructions pointed to by our tagged symbols needs to be rewritten because it contains unaligned free-branch instructions. This analysis produces a log of the tags corresponding to the instructions we need to modify. This log is consumed by the pre-processor during a second compilation phase in order to provide it with the previously missing information.

Unfortunately, inserting a `nop` at a certain position to fix a jump offset may actually affect the offsets of many other jumps since it alters the whole address space of the binary. Our prototype binary analyzer does not consider the overall structure of the binary file when reporting the instructions to fix. Therefore, while fixing a set of jump offsets, several other offsets may start to contain free-branch opcodes. This makes it necessary to perform several compilations until all the offsets are fixed. Note that in this process, we may need to fix a single jump instruction several times. However, since inserting `nop` instructions between a jump and its destination can only increase the offset but never decrease it, we are sure to find a safe offset after a finite number of iterations.

A more optimized analyzer that can perform a global analysis and take into account the target of every jump instruction would eliminate this problem. It would also produce smaller executables since recompilations insert otherwise unnecessary `nop` bytes.

5.2 Random Keys

As described in Section 4, our approach requires a random value to encrypt both the return address and the cookie stored on the stack. For this purpose, our prototype inserts a key generation routine at the beginning of the program's entry point (or initialization routine if it is a library). In our prototype, this routine simply reads a 32-bit random value from the Linux special file */dev/random* and stores the value in a global memory location.

If the attacker has a way to read arbitrary memory locations before performing the actual attack, he could be able to fetch the per-process random key and use it to craft the required values on the stack to defeat our implementation. This limitation is common to many canary-based stack protection mechanism such as StackGuard [11] and ProPolice [18]. However, this problem can be avoided by substituting the per-process random key with a per-function key computed at runtime in the function headers.

5.3 Stack Reference Adjustments

We store our cookie just above the saved return address in the stack, shifting the frame base upwards by 4 bytes. Since a function usually uses the `%ebp` register to reference the stack relative to the frame base, and our cookie is located below the frame base, references to the stack local variables remain unchanged. On the contrary, references to function parameters which are stored below the frame base, and therefore below our cookie, need to be adjusted by 4 bytes.

We achieve this by simply correcting each positive displacement to `%ebp` by adding to it the size of our cookie:

```
movl 0x8(%ebp), %eax   ⇒   movl 0xc(%ebp), %eax
```

Note that compiler optimizations that adopt Frame Pointer Omission (FPO) use the stack pointer to reference arguments and local variables. In this case, we need to compute the displacement of the stack pointer to the function's frame at any given position in the function in order to identify and fix the references and locate our cookie in the stack. This requires a comprehensive stack depth analysis. We have designed our pre-processor to perform this analysis on the fly without the need for any extra pass over the source file, even when the execution flow of the processed function is non-linear. We keep track of `push` & `pop` operations and arithmetic computations on the stack pointer and update the system's view of stack depth accordingly. Depending on the state of the stack, we can then decide whether a stack access (e.g., `120(%esp)`) points to a local variable or to a function's parameter, so that we can apply the displacement adjustment where appropriate.

5.4 Conditional Code Rewriting

Our prototype implements all immediate and displacement reconstruction strategies we described in Section 4. However, to reduce the performance overhead, we apply those transformations only when absolutely necessary. Otherwise, we use a faster approximate solution. In particular, during the first compilation phase, we prepend each instruction that contains free-branch opcodes among its immediate or displacement fields with an alignment sled. The sled protects the instruction, but does not actually remove the free branch from the code. Therefore, an attacker can sometimes build very short gadgets that fit the few bytes between the end of the sled and the unaligned free-branch instruction.

Our system automatically checks these bytes after the compilation. If it detects that they do indeed contain valid instructions, it falls back to the safer (but slightly less efficient) immediate or displacement reconstruction methods.

5.5 Return Address and Indirect Jump/Call Protection Blocks

As previously explained in Section 4, our solution protects aligned free-branch instructions by introducing two short blocks of code: the return address protection block and the indirect jump/call protection block (the current implementations are shown in Figure 3). These two pieces of code are the only ones in the final executable that can still contain gadgets and, therefore, they must be carefully designed to prevent any possible attack.

The return address protection code is 11 bytes long and all bytes are under our control, with the exception of the 4-byte address of the random key, which could change for each compiled program and for shared libraries at each relocation. To ensure that the code is safe to use, we need to prevent this value from containing potentially dangerous instructions. In our implementation, we control the least significant two bytes by automatically inserting appropriate alignment directives into the assembled code when defining the key storage location, ensuring that the address always ends with the innocuous "`0xf0 0x00`" sequence. In addition, according to the Linux process memory layout, the most significant address byte of the `.bss` section (where we store our random key) is limited in practice to `0x08` for regular ELF executables and `0xb*` for shared libraries [1]. Therefore, it encodes either a variation of a *load immediate into register* instruction (e.g., `mov $IMM, %reg`), or an `or` instruction between two 8-bit operands.

The indirect jump/call protection block is 19 bytes long and contains an additional 4-byte-long dynamic section, the per-function constant identifier we generate at compile time to compute the cookie. The example shown in Figure 3 (that uses a `0x0f0f1f76` function identifier) is entirely gadget-free because it contains no aligned or unaligned instruction sequences that would make it possible for an attacker to reach `jmp *%edx` without invalidating its contents. In fact, any logic/arithmetic operation that does not yield a result of zero (e.g., `incl %ebp`, unless `%ebp` overflows) clears the zero flag in the processor and prevents the use of the conditional jump `jz .+4` (this instruction only jumps if the zero flag is set in the processor). Consequently, the value inside `%edx` is cleared.

Different values of the function identifier could potentially introduce a new and useful gadget; but since these constants can be arbitrarily chosen and do not need to be kept secret, we can easily work around problematic cases. In order to minimize the risk in the first place, we use simple heuristics such as using bytes that represent invalid opcodes (e.g., `0x0f 0x0f`) and avoiding dangerous opcodes such as those encoding `mov` or free-branch instructions.

Figure 4 shows all the gadgets that can be extracted from our current system implementation. As can be seen, apart from the ability to load the `%eax` with a controlled value (`popl %eax`), the gadgets have no value.

5.6 Compiling `glibc`

During our case study of compiling `glibc` using G-Free, we have encountered several issues requiring particular care. These were mostly related to unconventional programming practices used for dealing with low-level tasks, or manually optimized assembly code. This section explains our observations in this regard, and explains how we cope with these special cases.

Multiple Entry Points: We have come across various functions in `glibc` that include more than one possible entry point. Our system

[1]The Linux process memory layout dictates that dynamic shared libraries are loaded at the address range `0xc0000000-0x40000000`, starting from higher addresses. As a result, in practice almost any shared library has `0xb*` as the most significant address byte of its `.bss` section.

Return address protection code

```
50              pushl   %eax
a1 00 f0 fd b7  movl    0xb7fdf000, %eax
31 44 24 04     xorl    %eax, 0x4(%esp)
58              popl    %eax
```

Indirect jump/call protection code

```
 50              pushl   %eax
 a1 00 f0 fd b7  movl    0xb7fdf000, %eax
 35 76 1f 0f 0f  xorl    $0x0f0f1f76, %eax
 39 45 04        cmpl    %eax, 0x4(%ebp)
 58              popl    %eax
 74 02           jz      freebranch
 31 d2           xorl    %edx, %edx
freebranch:
 ff e2           jmp     *%edx
```

Figure 3: Code inserted to protect the aligned return and indirect jump/call instructions

Gadget A.1

```
00 f0     addb   %dh, %al
fd        std
b7 31     movb           $0x31, %bh
44        incl           %esp
24 04     andb           $0x04, %al
58        popl           %eax
```

Gadget A.2

```
f0 fd     lock std
b7 31     movb   $0x31, %bh
44        incl   %esp
24 04     andb   $0x04, %al
58        popl   %eax
```

Gadget A.3

```
04 58   addb   $0x58, %al
```

Gadget B.1

```
 45      incl   %ebp
 04 58   addb   $0x58, %al
 74 02   jz     freebranch
 31 d2   xorl   %edx, %edx
freebranch:
 ff e2   jmp    *%edx
```

Figure 4: Gadgets available in the return address (A) and in the indirect jump/call (B) protection blocks

successfully detects such functions and instruments all entry points with the appropriate headers. Additionally, we prepend each header that lies in the execution path of other entry points with a jump instruction to skip over the header, ensuring that only one header is executed per function call.

Functions that Access the Saved Return Address: In `glibc`, we have encountered a single function, namely `setjmp` that accesses the saved return address on the stack. `setjmp`, together with the function `longjmp`, is used for implementing *non-local jumps*: a call to `setjmp` saves the current stack context to restore it afterwards when `longjmp` is invoked. This behavior conflicts with our return address protection scheme. Since the return address is stored in an encrypted form on the stack, a call to `setjmp` saves the encrypted return address and a subsequent call to `longjmp` results in an illegal memory access. In order to solve this problem, we modified our prototype to detect when the saved return address is moved to a register and perform the decryption on the duplicated value to ensure correct functionality.

Jumps between Functions: In numerous cases, a function directly jumps to another one without saving the return address, essentially making that jump an exit point. During compilation, we check every jump destination to recognize jumps outside the current function and treat them as regular exit points for inserting the necessary footers. These footers are not meant to protect a free-branch instruction, since none follows, but to restore the return address to its original value before transferring the execution flow to another function.

6. EVALUATION

The main goal of our evaluation is to show that our technique can be applied to compile real-world applications and produce gadget-free executables. To demonstrate that we are able accomplish this goal, we performed a set of experiments in which we measured the impact of our code transformations in terms of performance and size overheads of the binaries produced by our tool.

In our tests, we combined the G-Free pre-processor with `gas 2.20` and `GCC 4.4.3`. All the experiments were performed on a 2GHz Intel Core 2 Duo T7300 machine with 2GB of memory, running Arch Linux (i686) with Linux kernel 2.6.33.

6.1 Compilation Results

Since ROP attacks usually extract their gadgets from common libraries, we focus our evaluation on `glibc` version 2.11.1. The original version compiled without G-Free contains 9921 `ret` instructions (6106 of which unaligned) and 8018 `jmp*/call*` instructions (6602 of which unaligned). This sums up to almost 18K free-branch opcodes, each of which can be potentially used by an attacker to build many different gadgets.

After we compiled `glibc` using our system, all unintended `ret` and `jmp*/call*` instructions were either removed or made ineffective by prepending them with an alignment sled. In addition, all aligned free-branch instructions were protected by adding our return and indirect jump/call protection blocks. As a result, the library compiled with G-Free contained only the four type of gadgets we present in Figure 4.

However, due to the newly inserted code and instruction rewriting techniques, the size of the gadget-free version of the library increased by 30%. Although this value might appear to be high, one third of the overhead is caused by `nop` instructions included in the alignment sleds. As already discussed in Section 5, most of these could be eliminated by a more optimized implementation.

Unfortunately, providing a gadget-free version of `glibc` is not sufficient to completely prevent ROP attacks, since the attacker could still build the gadget set from other libraries or the application binary itself. Therefore, to achieve a complete protection

Program Name and Version	Original Size(KB)	G-Free Size (Overhead)	Unaligned `ret`	Unaligned `jmp*/call*`	Number of functions with RAP	Number of functions with JCP
glibc 2.11.1	1320.4	1728.4 (30.9%)	6106	6602	2817	827
gzip 1.4	72.7	92.4 (27.0%)	433	410	122	10
grep 2.6.3	86.3	106.3 (23.2%)	523	369	174	20
dd coreutils-8.5	48.0	57.9 (20.6%)	252	181	95	8
md5sum coreutils-8.5	30.9	37.7 (22.1%)	203	86	68	3
ssh-keygen openssh-5.5p1	140.6	182.5 (29.7%)	607	712	271	20
lame 3.98.3	322.6	406.6 (26.0%)	2228	1342	669	28

Table 2: Statistics on binaries compiled with G-Free (RAP=Return Address Protection, JCP=indirect Jump/Call Protection)

Program Name	Test case	Execution Time (seconds) Original Version	G-Free Version (Overhead)
gzip	Compress a 2GB file	66.5	68.4 (2.9%)
grep	Search in a 2GB file	81.3	82.9 (2.0%)
dd	Create a 2GB zero-filled file	86.6	88.9 (2.6%)
md5sum	Compute hash of a 2GB file	82.5	82.9 (0.6%)
ssh-keygen	Generate 100 2048-bit RSA keys	51.2	53.6 (4.6%)
lame	Encode a 10 min long wav file	115.5	122.0 (5.6%)

Table 3: Performance comparisons when the application and all the linked libraries are compiled with G-Free

against ROP, it is necessary to compile the entire application and all the linked libraries with our technique. To demonstrate that our tool can be applied to this more general scenario, we include in our evaluation a number of common Linux applications.

Table 2 shows statistics about the binaries compiled with G-Free. Our tool was able to successfully remove all unintended instructions and protect the aligned ones with an average size increase of 25.9% (more than half of which were caused by redundant `nop` instructions). The last two columns show that most of the functions can be protected by our very efficient return address encryption technique while very few of them required the more complex indirect jump/call protection block. This is a consequence of the fact that, according to what we observed in our experiments, programs rarely use `jmp*/call*` instructions.

6.2 Performance Measurements

Table 3 shows the performance overheads we measured by running the different applications compiled with our prototype (this includes the gadget-free versions of the programs and all their linked libraries). For each application, we designed a set of program-specific test cases, summarized in Column 2 of the table. The average performance overhead was 3.1% – a value comparable with the overhead caused by well known stack protection systems such as StackShield [35] and StackGuard [11].

Since a library cannot be run as a standalone program, we evaluated the performance overhead of our gadget-free version of `glibc` using a set of well-known benchmarks. In particular, we downloaded and installed the Phoronix Test Suite [1] which provides one of the most comprehensive benchmark sets for the Linux platform. Table 4 lists a sample of the benchmarks that represent different application categories such as games, mathematical and physical simulations, 3D rendering, disk and file system activities, compression, and well-known server applications. The results indicate that the performance overhead of an application using our gadget-free version of `glibc` is on average 1.09%.

7. CONCLUSIONS

Return-oriented programming is an attack technique that recently attracted significant attention from the scientific community. Even though much research has been conducted on the topic, no comprehensive defense mechanism has been proposed to date.

In this paper we propose a novel, comprehensive solution to defend against return-oriented programming by removing all gadgets from a program binary at compile-time. Our approach targets all possible free-branch instructions, and, therefore, is independent from the techniques used to link the gadgets together. We implemented our solution in a prototype that we call G-Free, a preprocessor for the `GNU Assembler`. Our experiments show that G-Free is able to remove all gadgets at the cost of a very low performance overhead and an acceptable increase in the file size.

8. ACKNOWLEDGMENTS

The research leading to these results was partially funded from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement nř 257007. This work has also been supported in part by the European Commission through project IST-216026-WOMBAT funded under the 7th framework program. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein. We would also like to thank Secure Business Austria for their support for this research.

9. REFERENCES

[1] Phoronix test suite. `http://www.phoronix-test-suite.com/`.

[2] Rop attack against data execution prevention technology, 2009. `http://www.h-online.com/security/news/item/Exploit-s-new-technology-trick-%dodges-memory-protection-959253.html`.

[3] Symantec: Internet Security Threat Report. `http://www4.symantec.com/Vrt/wl?tu_id=jLac123913792490340803`, 2009.

[4] Intel 64 and IA-32 Architectures Software Developer's Manuals. `http://www.intel.com/products/processor/manuals/`, 2010.

[5] Aleph One. Smashing the stack for fun and profit. In *Phrack Magazine n.49*, 1996.

[6] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[7] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham. Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage.

Benchmark	Orig. Libc	G-Free (Overhead)
FS-Mark (Files/s)	15.1	14.9 (1.3%)
IOzone-write (MB/s)	22.8	22.6 (0.4%)
IOzone-read (MB/s)	23.0	22.7 (1.4%)
Dbench (MB/s)	83.7	82.0 (2.0%)
Minion (s)	250.2	250.7 (0.2%)
Sudokut (s)	97.1	100.4 (3.5%)
TSCP (Nodes/s)	224642.0	224385.0 (0.1%)
GMPbench (Score)	2955.5	2954.5 (0.03%)
BYTE (Lines/s)	7288371.3	6948792.8 (4.6%)
PyBench (s)	6791.0	6959.0 (2.5%)
PHP Comp (s)	102.9	107.3 (4.3%)
7-Zip (MIPS)	2822.0	2802 (0.7%)
Unpack Linux Kernel (s)	30.30	31.01 (2.3%)
LZMA (s)	291.67	291.86 (0.01%)
BZIP2 (s)	65.63	65.84 (0.3%)
FLAC Audio Encoding (s)	12.96	13.09 (1.0%)
Ogg Encoding (s)	27.14	27.20 (0.2%)
Himeno (MFLOPS)	152	151.44 (0.4%)
dcraw (s)	52.68	52.99 (0.6%)
Bullet Physics Engine (s)	39.58	39.74 (0.4%)
Timed MAFFT (s)	52.48	52.55 (0.1%)
PostgreSQL (Trans/s)	155.24	156.66 (0.9%)
SQLite (s)	189.09	191.78 (1.4%)
Apache(Requests/s)	7129.05	6836.24 (4.1%)
x2642009 (Frames/s)	13.72	13.62 (0.7%)
GtkPerf (s)	20.89	20.49 (1.9%)
x11perf (Operations/s)	912000	912000 (0.0%)
Urban Terror (Frames/s)	34.20	34.05 (0.9%)
OpenArena (Frames/s)	46.93	46.67 (0.6%)
C-Ray (s)	553.7	554.0 (0.05%)
FFmpeg (s)	24.93	25.02 (0.4%)
GraphicsMagick (Iter/min)	45	44 (2.2%)
OpenSSL (Signs/s)	25.28	25.28 (0.0%)
Gcrypt Library (micros)	6963	6983 (0.3%)
John The Ripper (Real C/S)	1854667	1857333 (0.1%)
GnuPG (s)	20.46	20.67 (1.0%)
Timed HMMer Search (s)	88.93	89.31 (0.4%)
Bwfirt (s)	284.9	285.3 (0.2%)
Average:		1.09%
Std:		1.27

Table 4: Performance comparison of the original and G-Free `glibc` using benchmarks from the Phoronix Test Suite

In *Proceedings of EVT/WOTE 2009. USENIX/ACCURATE/IAVoSS*, 2009.

[8] S. Checkoway and H. Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical report, 2010.

[9] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *Lecture Notes in Computer Science*, 2009.

[10] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th Usenix Security Symposium,*, 2003.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7^{th} USENIX Security Symposium, USA*, 1998.

[12] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition,*, 2000.

[13] L. Davi, A. R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *Proceedings ACM workshop on Scalable trusted computing*, 2009.

[14] Felix Lidner. Confidence 2.0,. Developments in Cisco IOS forensics.

[15] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of CCS*, 2008.

[16] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, 2008.

[17] M. Frantsen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proceedings of USENIX security*, 2001.

[18] Hiroaki Etoh. GCC Extension for Protecting Applications from Stack-Smashing Attacks (ProPolice). In *http://www.trl.ibm.com/projects/security/ssp/*, 2003.

[19] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium, USA*, 2009.

[20] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[21] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits with Return-less Kernels. In *Proceedings of the 5th ACM SIGOPS EuroSys Conference*, 2010.

[22] M. W. Lucas Davi, Ahmad-Reza Sadeghi. Ropdefender: A detection tool to defend against return-oriented programming attacks. Technical report, Technical Report HGI-TR-2010-001.

[23] Nergal. The advanced return-into-lib(c) exploits. In *Phrack Magazine n.58*, 2001.

[24] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25^{th} Annual Computer Security Applications Conference (ACSAC), Honolulu, Hawaii, USA.*, pages 60–69. IEEE Computer Society, Dec. 2009.

[26] Scut, Team Teso. Exploiting format string vulnerabilities. 2001.

[27] Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005. `http://www.suse.de/~krahmer/no-nx.pdf`.

[28] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of Operating System Symposium SOSP*, 2007.

[29] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, 2007.

[30] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CSS)*, 2004.

[31] Solar Designer. return-to-libc attack. Technical report, bugtraq, 1997.

[32] The PaX Team. Pax address space layout randomization. Technical report, http://pax.grsecurity.net/docs/aslr.txt.

[33] The PaX Team. Pax non-executable pages. Technical report, http://pax.grsecurity.net/docs/noexec.txt.

[34] Tim Kornau. Return oriented programming for the arm architecture. Technical report, Master's thesis, Ruhr-Universität Bochum, 2010.

[35] Vendicator. Stackshield: A "stack smashing" technique protection tool for linux. Technical report, http://www.angelfire.com/sk/stackshield/.

[36] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS*, 2009.