Defeating Return-Oriented Programming Using Compiler-Assisted Fine Grained Load-Time Randomization

Michael Brengel
Saarland University
mbrengel@mmci.uni-saarland.de

Vikram Narayanan
Saarland University
s8vinara@stud.uni-saarland.de

Sebastian Miga
Saarland University
s8dimiga@stud.uni-saarland.de

Omer Asif
Saarland University
s9omasif@stud.uni-saarland.de

Abstract

In this paper we present a novel compiler-assisted approach for protecting a program against return-oriented programming (ROP) attacks. During compile time we increase the code size of each function of the program by adding no-op instructions after each function epilogue. By modifying the binary loader of the operating system we can then randomly shift the instructions inside the function bodies, thereby completely randomizing the binary without breaking the semantics of the program. Since we leave the function addresses unchanged for simplicity reasons, we also propose a simplistic mechanism to protect against entry-point gadgets, which would otherwise still be valid attack vectors. With those two approaches in tandem, we significantly decrease the chances of an attacker successfully crafting a working gadget chain.

1 Introduction

In today's computing world, despite a decade-long research effort and many proposed countermeasures, runtime attacks which aim at subverting the control flow of a program are still the main source of vulnerabilities with return-oriented programming (ROP) [14, 22] being the strongest kind of such an attack. A ROP attack is a generic code-reuse attack where the attacker gains control over the stack and abuses this control to execute known small instruction snippets of the target binary, so called gadgets. By linking together those gadgets in the right order, the attacker can, in principle, achieve turing completeness[22]. Although this concept seems to be impracticable at a first glance, ROP attacks have been in the past, and still are a severe threat for real-world applications. For instance, the Safari Browser from Apple, was vulnerable to a ROP attack which was used to jailbreak the iOS operating system with a sophisticated gadget chain containing 150 API calls [4].

Early protection mechanisms against ROP like address space layout randomization (ASLR) [1] and stack smashing protection (SSP) [12] did significantly raise the bar for attackers. However, information leakage and guessing attacks have enabled them to bypass these protection mechanisms [24, 25]. Therefore, a lot of research effort went into developing new protection mechanisms to further secure applications from ROP attacks, each of which either has strong limitations or was shown to be incomplete to some extent.

We give a small overview[1] of such mechanisms to motivate our own approach. A simple compiler-assisted protection proposed by [19] is to alter the compiler such that it avoids emitting instructions which can later be used by the attacker as return instructions. However, [10] has shown that ROP attacks do not necessarily require return instructions for a working exploit, which makes this defense incomplete. Other approaches rely on dynamic analysis, as ROPdefender [13] for example, which monitors the execution trace during runtime and keeps a shadow stack, which can be used to detect malicious behavior caused by ROP attacks and terminate the program if it encounters one. These approaches usually suffer from a noticeable overhead, which most developers are not willing to take. Even control-flow integrity (CFI) [7], a strong comprehensive approach, which aims at validating the integrity of the caller, has lately been bypassed [18]. CFI also partially suffers from overhead in the sense that it can give stronger security guarantees, but this in turn would increase the overhead.

Another type of protection against ROP is to randomize the binary. Compared to the other approaches, randomization aims at giving probabilistic security guarantees by randomly modifying the program at compile-time and/or run time. Our approach uses such a technique to propose a defense mechanism against ROP.

We propose a novel compiler-assisted approach cou-

[1] This overview is by no means complete.

pled with load-time randomization. The approach works by modifying a compiler such that it increases the code size of the functions. As a next step, we add the addresses of the modified functions to the binary in the form of a new section. Then we modify the binary loader of the operating system such that if it detects this section inside a binary, it randomly shifts the instructions inside the function bodies, which in turn implies that the attacker does not know the location of the gadgets. Since the function addresses are not modified by the loader, an attacker could still use whole functions as gadgets. Therefore, we add protection mechanisms for these kind of gadgets to the compiler as well.

The remainder of this paper is structured as follows: In section 2 we give a brief overview of ROP attacks. In section 3 we describe the attacker model. Section 4 gives a detailed explanation of our approach. Followed by section 5, where we discuss the implementation in the LLVM compiler framework and the Linux ELF loader. We evaluate our approach by randomizing the GNU make binary and analyze it concerning correctness, code size and performance under section 6. In section 8 we discuss limitations and future work regarding the implementation followed by the conclusion in section 9.

2 Background

In this section we give a short introduction to ROP attacks on x86-64 systems. In order to mount a ROP attack, the attacker first needs to gain control over the stack, which usually happens due to a buffer overflow vulnerability. By modifying the stack, the attacker can then alter the execution flow of the program. For instance, the attacker can modify the return address of a function, which is stored on the stack.

Consider for example the case where an attacker has gained control over the stack, modified it and the vulnerable function is about to execute the `ret` instruction and the stack looks as depicted in Figure 1. The `ret` instruction will pop the element on top of the stack and change the instruction pointer to that value. In our case, the top element of the stack is the address of `gadget1`[2]. The instructions of `gadget1` will pop the address of the `"bash"` string into the `rdi` register. The next `ret` instruction will then change the instruction pointer to the value on top of the stack, which happens to be the address of the `system` function in our case. The `system` function assumes its first argument to be located in the `rdi` register[3], which points to the `"bash"` string. The attacker now has a working bash prompt that he could use to further

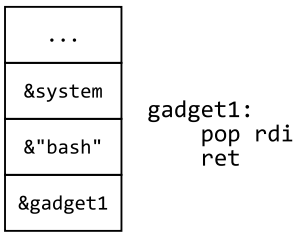

Figure 1: An example of a stack during a ROP attack.

```
<foo>:
<0> : 55               push  rbp
<1> : 48 89 e5         mov   rbp,rsp
<4> : b0 01            mov   al,0x1
<6> : 3a c3            cmp   al,bl
<8> : e8 00 4b 00 00   call  0x4b00
<13>: 48 31 c0         xor   rax,rax
<16>: 5d               pop   rbp
<17>: c3               ret
```

Figure 2: An example function containing several gadgets.

compromise the system. A command prompt might be useful on systems where the attacker has only a limited view, for example on remote systems. But, also on local systems, such an attack could make sense if the setuid bit of the target binary is set and the binary is owned by a user with higher privileges. However, it should be noted that the attacker can enforce any execution by providing the right set of gadgets which goes far beyond spawning just a shell.

The power and effectiveness of ROP are due to numerous reasons. Firstly, ROP completely bypasses data execution prevention (DEP) [8] since gadgets can be chosen from the text section which is guaranteed to be executable.

Secondly, although it seems impossible to construct an effective gadget chain, it is feasible in practice. Tools are available which can automate the process of locating gadgets and constructing gadget chains[3]. Those tools benefit from the fact that programs often use a large variety of (shared) libraries, which significantly increase the code area which in turn increases the likelihood of finding useful gadgets. On x86 systems, the attacker can also exploit the fact that instructions are variable-sized, which increases the number of potential gadgets. We distinguish between three types of gadgets, which we illustrate with the function depicted in Figure 2. Those three types of gadgets are:

Aligned Gadgets Aligned gadgets are gadgets which start at intended instruction offsets. For example, if we start the gadget at offset 16, we get a `pop rbp`

[2]Note that the stack is upside down in memory.

[3]This is only true for x86-64. In the case of 32-bit, the attacker could simply put the address of `"bash"` on top of the address of `"system"` on the stack.

gadget[4]. If we start the gadget at offset 13, we get a `xor rax,rax; pop rbp` gadget, and so on.

Unaligned Gadgets As opposed to aligned gadgets, unaligned gadgets start at unintended offsets, which basically means that we jump "in the middle" of an instruction, which is a valid behavior on the x86 architecture. For example if we start the execution at offset 5, the following instructions will be executed:

```
01 3a      add  [rdx],edi
c3         ret
```

This gives us a `add[rdx],edi` gadget.

Entry-Point Gadgets Entry-Point Gadgets are a special case of aligned gadgets where the gadget starts executing at offset 0, i.e. the whole function `foo` is used as a gadget. Entry-Point Gadgets have lately been used to bypass CFI implementations [18].

Together, those three types of gadgets form the whole set of gadgets.

3 Threat Model

We assume a scenario where a binary is being executed and a function is getting some input provided by the attacker. This does not necessarily mean that the binary is run by the attacker. Consider, for example, the case where a PDF reader has a vulnerability and the attacker only provides a malformed PDF file which exploits this vulnerability and the program is being executed by a victim.

We make the following assumptions regarding the attacker's capabilities:

Binary Knowledge The attacker has full knowledge of the binary and its dependent libraries.

Full Stack Control The attacker managed to gain full control over the stack by exploiting a buffer overflow vulnerability, for example. We do not assume any limits on the number of bytes on the stack that can be modified by the attacker.

ASLR Bypass The attacker has managed to fully bypass address space layout randomization. We assume the strongest kind of such a bypass where the attacker knows the exact base pointers of all loadable segments.

No Full Memory Disclosure The attacker cannot get a full snapshot of the program loaded in memory. Since our approach randomly shifts the body of functions, this assumption is crucial.

[4]We do not include the `ret` instruction in this notation or any other instructions which transfer the control flow back to the gadget chain.

As for other probabilistic approaches, the purpose of our method is *not* to eliminate gadgets. Rather, we want to significantly decrease the chances of an attacker successfully crafting a working gadget chain. We will show that we can give probabilistic security guarantees against all three types of gadgets defined in section 2.

4 Approach / Design

Our approach is based on randomizing the entire binary, provided we have the source-code of the program, thus raising the bar for an attacker to guess where a particular gadget could be. The main objective of this approach is to randomize the position of gadgets present in a program/library without breaking the semantics of the actual code.

Return oriented programming relies on carefully constructing the gadget chain in order to achieve a particular functionality. With the approach of randomizing the function body of every function during every load, we make it extremely difficult for the attacker to guess where a particular code snippet would be at any given time.

In-place code randomization[20] offers a very enticing feature of not requiring one to have the sources of the binary or library. However, In-place code modifications suffer from several weaknesses.

Why not In-place? As discussed by the authors of the In-place code randomization paper, it is hard to give a definitive answer to whether or not the gadgets that could not be randomized are sufficient for constructing useful ROP code. The number of gadgets that can be randomized using the In-place approach for a set of PE files is 76.9%. The remaining percentage of code are un-modified by the In-place randomization approach. In contrast, our approach guarantees to randomize 100% of the code in memory.

Correlating both of the above, one could infer that it may be possible for an attacker to construct a valid gadget chain with the remaining amount of non-randomized gadgets.

Function address randomization ROP can be defended theoretically by placing each function at random places, preventing the attacker from constructing a valid gadget chain. However, this is not a pragmatic solution as all `jmp` and `call` instructions needs to modified on load time. This is complex and equally error prone.

As discussed in section 2, many types of gadgets are possible with this simple code snippet. By constructing a chain of such gadgets, an attacker could run a shell or gain access privileges on the victim's system.

```
<foo>:
<0> : 55              push  rbp
<1> : 48 89 e5        mov   rbp,rsp
<4> : b0 01           mov   al,0x1
<6> : 3a c3           cmp   al,bl
<8> : e8 00 4b 00 00  call  0x4b00
<13>: 48 31 c0        xor   rax,rax
<16>: 5d              pop   rbp
<17>: c3              ret
```

```
<foo>:
<0> : 49 81 fc 00 04 00 00   cmp r12, <cookie_value>
<7> : 75 00                  jne <real_func_offset>
<9> : 0f 0f 0f 0f 0f         (bad)
<14>: 55                     push  rbp
<15>: 48 89 e5               mov   rbp,rsp
<18>: b0 01                  mov   al,0x1
<20>: 3a c3                  cmp   al,bl
<22>: 49 c7 c4 00 00 00 00   mov   r12,0x0
<29>: e8 00 4b 00 00         call  0x4b00
<34>: 49 c7 c4 00 00 00 00   mov   r12,0x0
<41>: 48 31 c0               xor   rax,rax
<44>: 5d                     pop   rbp
<45>: 49 c7 c4 00 00 00 00   mov   r12,<cookie_value>
<52>: c3                     ret
```

Figure 3: A sample function before and after randomization.

The attacker relies on the knowledge of a particular binary and its gadgets to construct a gadget chain to achieve the necessary functionality. This is possible for the attacker as the function addresses remains fixed except for a fixed offset imposed by the ASLR (address space layout randomization).

We present a novel idea on randomizing every function in the binary and its associated libraries. Contrary to the idea of changing the function addresses, our approach randomizes the actual function body, thereby offering complete protection against all kinds of gadgets. Our approach adds additional details during the compilation phase and uses the added information to randomize the binary during load time. The following steps briefly describe what is done in each of these phases. For implementation details, refer to the implementation section 5.

Compile time The compiler is modified such that, during the code generation phase a number of `nop's` (say x) are injected into every function, thereby increasing the function size. Once the function is expanded, one could imagine that the function body is floating in a pool of `nops`.

Load time During the load time, the body of the function is slid inside the pool of `nops` by choosing a random number $n \bmod x$. The initial few bytes of every function will be patched with a `jmp` instruction to make the control flow jump to the actual body once the sliding is done.

Relative Addressing The X86-64 Application Binary Interface [6] explains the use of different code models depending on the requirements. The machine code generated by the compiler uses a certain kind of memory addressing model to access other code and data sections present in memory.

References to both code and data on x86-64 are done with RIP (instruction pointer) relative addressing. There are three types of code model, namely *small*, *medium* and *large*.

Small code model The small code model addresses all the code and data section in RIP relative addressing. The model requires all the symbols to be located in the virtual address range from 0 to $2^{31} - 2^{24} - 1$. This is the fastest code model and the one used by the compiler by default for all the programs.

Medium code model In the medium model, the data section is split into two parts — the data section still limited in the same way as in the small code model and the large data section having no limits except for available addressing space.

Large code model This is a completely relaxed model and does not make any assumptions on the code and data size.

Small PIC Dynamic libraries do not know the virtual addresses until it is dynamically linked. So all the addresses have to relative to the instruction pointer when the code is compiled with `-fpic` flag.

The medium and large code model require the compiler to use the `movabs` instruction to address code and data using absolute addressing. Since we randomize the function body, we need to patch up every addressed item if it is a relatively addressed. This is often exhaustive as even the normal calls are addressed using relative addresses in the small code model. We resort to the large code model making it easier for the loader to patch less number of offsets.

With these tools at our disposal, we randomize both the aligned and unaligned gadgets. However, there exists a possibility for the attacker to use entry-point gadgets. Entry-point gadgets can also be protected by disallowing a control transfer from a `ret` instruction of a function to the beginning of another function. We use a cookie like mechanism to keep track of function entry and function returns. When the function returns, it carries a value in a register reserved for this purpose. On each entry point, this cookie value is checked. If these two matches, there is a control flow from a return of a function to this entry point. To allow valid calls, we update the reserved register before every call instruction to `zero`.

During the load time, the cookie value is randomized for each and every program and remains the same until the program is unloaded from memory. The `nops` are replaced with illegal instructions during the load time, which would cause the program to halt if an attack code guesses the position of gadgets and jumps to that position. A sample function before and after randomization is illustrated in Figure 3. As the solution aims to protect against all kinds of gadgets, it is necessary to compile all the depending libraries using our randomization approach to ensure complete protection.

5 Implementation

We have implemented our approach for x86-64 Linux based systems. In the following paragraphs we will outline the technical details of the implementation and explain how the process of randomizing a binary with our method is carried out. Figure 4 describes the control flow of a program being randomized by our approach from source-code to the randomized image on load time. The source-code is compiled with the LLVM compiler framework which we have modified to apply our transformations to the binary. The resulting ELF file is then analyzed and modified by the section adder, which is a small python script that collects relevant information for randomizing the binary and adds that information to the binary in the form of a new section. The modified ELF loader checks each file for this randomization section and, if present, randomizes the binary on load time by shifting the offsets of the instructions. We will now describe each component in detail.

Low Level Virtual Machine (LLVM) We add our transformations to the LLVM compiler framework by implementing a machine function pass. This pass is executed for every function of the program and gives us enough flexibility to apply our transformations. The main tasks of the function pass are to increase the code size and to provide entry-point gadget protection. Increasing the code size is simply accomplished by adding a fixed number of `nop` instructions after every function epilogue[5]. To protect against entry-point gadgets, we apply three simple transformations. Firstly, at the beginning of each function we add the following instructions:

```
cmp r12, cookie_value
jne offset
```

The `cookie_value` value will be patched on load time as well as the value for `offset`, since it is important for security reasons that these values are randomized on each run. Secondly, each `ret` instruction is transformed to:

```
mov r12, cookie_value
ret
```

Finally, each `call foo` instruction is transformed to:

```
xor r12, r12
call foo
xor r12, r12
```

As we can see, the `r12` register is used to validate the integrity of the caller as explained in section 4. In order to not break the semantics of the program we had to reserve the `r12` register in LLVM. This might be a problem if a program contains inline assembly which uses the `r12` register (see section 8 for further details). We also enable the large code model by default for reasons explained in section 4.It is also worth noting that we disabled the usage of jump tables as they contain absolute addresses which point to the middle of functions which obviously breaks the semantics of the program if we plan to change the offsets of the instructions on load time. Currently, we also do not support position independent code (PIC), which disables the usage of shared objects. Hence, all libraries the program depends on must be linked statically.

Section Adder The section adder is a small python script which uses the `nm` and `objdump` utilities to analyze the compiled binary. It scans for the function addresses of the functions which have been modified by the compiler and adds those addresses to the ELF file in the form of a new section. Additionally it scans for `ret` instructions and adds the offsets of those instructions to the section as well as the cookie value needs to be patched on load time as well as explained in section 4. It should be noted that this could also be implemented by modifying a linker such as the GNU `ld` linker which is widely used. However, by offloading this task to a separate script we give the developer the freedom to use the linker of his choice.

[5] Currently, we add 20 `nop` instructions.

Figure 4: Life of a program being compiled, modified and randomized on load time.

ELF Loader We present the changes to the Linux ELF loader present in the kernel to achieve the functionality of load time randomization.

When a program is invoked in the user-space, the "exec" system call transfers the control to the kernel to load the particular program. The kernel identifies the binary format and calls the relevant handler registered for handling the loading of that particular format (in our case ELF). The binary handler basically reads the binary and identifies the "loadable" section and maps it into the memory. Once the data is loaded into the memory and the permissions are set, the kernel updates many data structures for its book-keeping and eventually the control is transferred back to the user space. The program now has its own virtual space to run on.

We have modified the file `fs/binfmt_elf.c` of the Linux kernel, which is responsible for handling the ELF file formats. Similarly, it can be extended to support other file formats as well. The binary handler parses the sections of the ELF binary and checks for the special section added by our section adder. If present, it reads the data from the added section.

The special section data consists of offsets of all functions in the binary, the offsets where the cookies have to be patched up for entry-point gadget protection. It uses the kernel's random number generator to generate random numbers. One is for the cookie, which remains constant for one load instance of the program. The other random number is for the random shift offset and this changes for every function. This ensures that every function is slided in a completely random fashion. Once the function body is moved/slided to the new location, the jump address in the function prologue is patched up to jump to the actual function body in case of a valid function entry.

The loader goes through each function offset and patches the cookie, slides the body around and the `nops` are patched to illegal instructions that makes the random guessing jump impossible for the attacker. The kernel is then left to follow its normal workflow for loading the program.

Though our solution is Linux centric, it is fairly straight-forward to achieve the same functionality in any other loader.

6 Evaluation

Our approach impacts performance due to additional instructions added to each function in order to provide gadget randomization. These consist in a comparison instruction between the contents of `r12` and a randomly generated value which is earlier referred to as the cookie and used to avoid entry point gadgets, a `jmp` to the actual function body over an arbitrarily long sequence of illegal instructions(in our implementation at most 20 priviledged instruction bytes), a `cmp` and `ret` instruction in order to check the cookie's integrity as well as a `nop` sequence in the epilogue that allows us to move the respective's function body without breaking its semantics.

Due to the fact that all functions within a binary are altered the overall size and performance are directly affected by the number of functions within the given binary. The results of compiling the linux kernel using two versions of the make utility are presented in Figure 3.1. The standard make utility is the default binary shipped with most Linux distributions while the other, was compiled using our approach and ran on a system running our customized ELF loader.

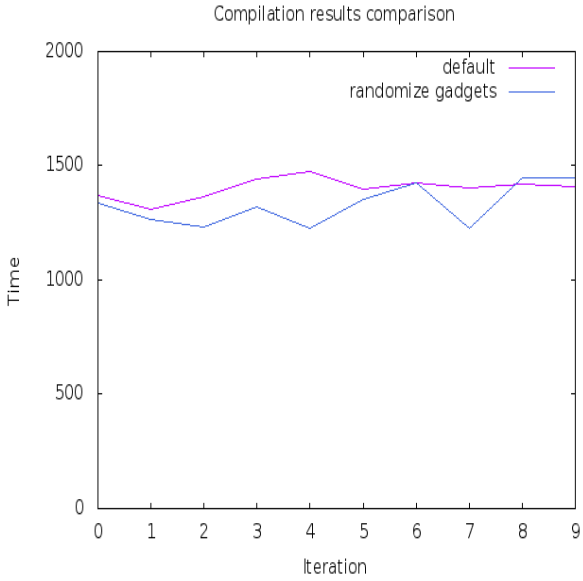

Figure 3.1 The graph shows a comparison between the performances obtained using the default version of the make binary as well as the one implementing our approach

7 Related Work

Currently, there are numerous defense mechanisms that aim to detect memory exploits which lay at the core of mounting ROP attacks. Compile-time solutions such as ProPolice[15] and StackGuard[17] try to detect stack overflows. PointGuard[9] prevents pointer corruption by employing encryption over pointers stored in memory.

Approaches such as StackGhost[16] and StackShield[5] make use of a shadow stack that stores all return addresses and checks for alterations on function exits. Thus ROP chaining is prevented in case of return address corruption. Address Space Layout Randomization(ASLR)[1] is an effective technique that prevents ROP attacks. This technique randomizes positions of code segments, stack and heap as well as the base address of dynamically linked libraries within the address space of a process. As a result, the attacker must guess the correct addresses where the gadgets are located in order to successfully mount a ROP attack. Although ASLR raises the bar for attackers it has been shown that due to the limited entropy provided by existing implementations evasion is possible by either exploiting the Global Offset Table and retrieving target function addresses [21] or even performing a brute-force attack on 32-bit applications[23].

The method described in [11] relies on the observation of instruction sequences in order to detect ROP-based attacks. A gadget is interpreted as being a short sequence of instructions that end with a `ret` instruction. Dynamic binary instrumentation is used in order to count the number of instructions between two consecutive `ret` instructions. A predefined threshold determines if an alert is risen depending on the number of consecutive instructions ending with a `ret`: if there are at least three consecutive successions of five or less instructions that end with a `ret` opcode. The approach inherently suffers from the fact that a heuristic is used to detect ROP attacks. Another downside is that it can be bypassed by avoiding `ret` instructions [19].

A solution similar to our own is presented in [20]. It aims to remove gadgets by in-place binary alteration. This choice is motivated by the lack of debugging and relocation information which in turn, provides incomplete disassembly coverage. In order to remove ret gadgets the approach implements atomic instruction substitution where a set of instructions is replaced with another, functionally equivalent and of the same size, instruction reordering which removes gadgets by reordering a set of instructions so that the functionality is not changed and register reassignment where a set of registers may be substituted with another in a sequence of instructions in which all registers in that set are live. This solution does not remove all gadgets due to incomplete disassembly coverage.

8 Discussion and Limitations

Compiler-assisted code modification, as the name suggests, requires the source-code in the first place. This may not always be possible as there are third-party applications whose sources are not open. For instance, the Adobe PDF reader has suffered from a lot of vulnerabilities [2] which cannot gain from this defense mechanism. Even with the source-code, this solution requires all its dependent libraries to be compiled with the same approach to completely randomize the binary and its associated libraries.

Also, to support dynamic libraries, we need to modify the user space loader component (`/lib64/ld-linux-x86-64.so.2` in our system), which is a part of the glibc. This forces one to use only statically linked binaries, which obviously increases the binary size. We leave this as a future work.

As our approach is based on inserting the no-ops and randomly sliding the code within that pool of no-ops, the system loader needs to be modified. Modification to the kernel code makes it cumbersome when someone wants to update their kernel. This calls for a recompilation of the new kernel with our loader modifications.

The machine-code generated by the compilers can possibly be in any of the three code-models namely, small, medium and large. We decided to take the large code-model for our prototyping as the small and medium code model makes all the function calls as `rip`-relative. Though, patching up all the relative offsets feasible, it is complex and error prone. We leave this as a future work.

System components and low-level software often contain hand-written inline assembly snippets. As our approach reserves the `r12` register in order to provide a defense against entry-point gadgets, using the same register in the inline assembly might create unnecessary problems as `r12` is totally in control in all the remaining parts of the code. As a future work, we also would like to automatically take care of this during the compilation phase.

The usage of the `r12` register for entry-point gadget protection leaves our code x64 dependent. However, x86 can also be supported by using the `ebp` register instead of `r12`. To do this, we need to instruct the compiler not to use the base pointer for accessing the function arguments.

9 Conclusion

In this paper we have presented a novel approach of defending against ROP based attacks by randomizing the gadgets at load-time. Our approach aims to defend

against all kind of gadgets explained in the 3. By not relying on in-place modifications, we are able to randomize 100% of the binary provided that the source code of the binary and all dependent libraries are available.

References

[1] Address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[2] Adobe reader CVE details. http://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html?vendor_id=53.

[3] Ropgadget - gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/.

[4] Technical analysis on iphone jailbreaking. http://community.websense.com/blogs/securitylabs/archive/2010/08/06/technical-analysis-on-iphone-jailbreaking.aspx.

[5] Vendicator. stackshield: A stack smashing technique protection for linux. http://www.angelfire.com/sk/stackshield.

[6] X64 ABI application binary interface. http://www.x86-64.org/documentation_folder/abi.pdf.

[7] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications.

[8] ANDERSEN, S., WRITER, T., AND ABELLA, V. Memory protection technologies. https://technet.microsoft.com/en-us/library/bb457155.aspx.

[9] C. COWAN, S. BEATTIE, J. J., AND WAGLE, P. Pointguard:protecting pointers from buffer overflow vulnerabilities. in proceedings of the 12th usenix security symposium.

[10] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns.

[11] CHEN, P., XIAO, H., SHEN, X., YIN, X., MAO, B., AND X, L. Drop:detecting return-oriented programming malicious code.

[12] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.

[13] DAVI, L., SADEGHI, A., AND WINANDY, M. Ropdefender: a detection tool to defend against return-oriented programming attacks.

[14] DESIGNER, S. Getting around non-executable stack (and fix).

[15] ETOH, H. Gcc extension for protecting applications from stack-smashing attacks (propolice). in http://www.trl.ibm.com/projects/security/ssp/.

[16] FRANTSEN, M., AND SHUE, M. Stackghost: Hardware facilitated stack protection.

[17] FRANTSEN, M., AND SHUEY, M. Hardware facilitated stack protection. in proceedings of usenix security.

[18] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity.

[19] LI, J., WANG, Z., JIANG, X., GRACE, M., AND BAHRAM, S. Defeating return-oriented rootkits with "return-less" kernels.

[20] PAPPAS, V., POLYCHRONAKIS, M., AND KEROMYTIS, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization.

[21] ROGLIA, G. F., MARTIGNONI, L., PALEARI, R., AND BRUSCH, D. Surgically returning to randomized lib(c). in proceedings of the 25th annual computer security applications conference (acsac), honolulu, hawaii, usa, pages 6069. ieee computer society.

[22] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86).

[23] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization.

[24] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A.-R. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization.

[25] STRACKX, R., YOUNAN, Y., PHILIPPAERTS, P., PIESSENS, F., LACHMUND, S., AND WALTER, T. Breaking the memory secrecy assumption.