Concordia Institute for Information System Engineering (CIISE)
Concordia University

INSE 6140 Malware Defenses and Application Security

Project:

A Survey on Return Oriented Programming

Submitted to:

Professor Dr. Makan Pourzandi

Submitted by:

Student name	Student ID
Guillaume Pillot	40023833

Date

15 April 2016

1. Introduction

Since the first buffer overflow exploitation in the end of 80's, many protections mechanism have been developed. Data Execution Prevention (DEP) or Non-executable section memory (NX), it is not possible to execute malicious code in the stack that an attacker would have injected like a shellcode. Address Space Layout Randomization (ASLR) places the segments memory of the program (stack,heap and shared library) in a random location of the memory making buffer-overflow attacks hardier.

The Return Oriented Programming (ROP) allows to circumvent these protections using the code, already in the program. It is easy to find a part of code in a fixed location in the memory and that can be executable in order to bypass the DEP and ASLR mechanisms.

2. Basics

ROP basics employ an assembly instructions list that ends by the `ret` instruction named "gadget". This instruction takes the saved instruction pointer on the stack and upgrades `esp` and `eip`. So, the attacker can overwrite the return address on the stack (`ebp` +4) and redirect the execution flow. The attacker can chain the gadgets execution thanks to the `ret` instruction that allows to keep the execution flow control. The ROP attack is Turing-complete, so the attacker can execute any algorithms.

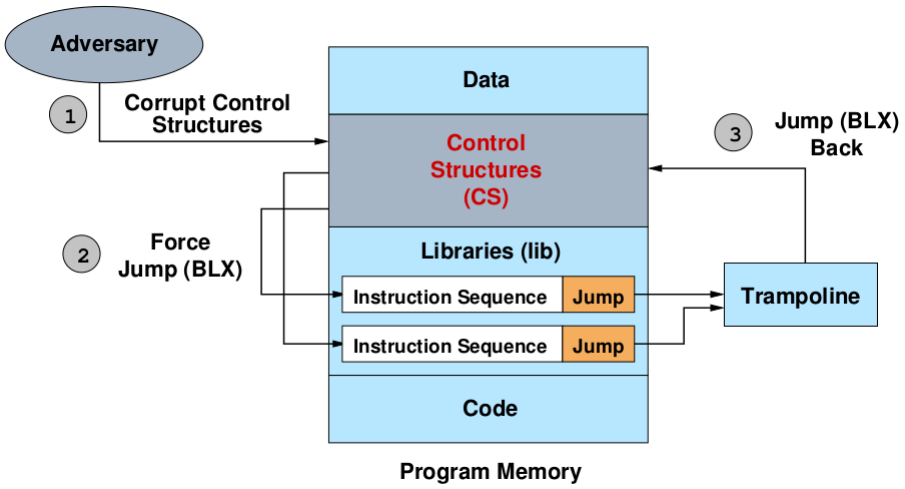
But the first goal of attackers aims to disable the DEP protection. For example, on Windows, an attacker can call the API's function "SetProcessDEPPolicy" and then disactivate the DEP mechanism so that he can execute again a shellcode on the stack.

Many solutions have been developed for detecting ROP attacks as for example detecting a anomalous usage of `ret` instruction or deleting `ret` opcode. Unfortunately, most solutions are not complete or need information about the program like source code or debug information. These protections can be categorized, each of them with its flaw.

3. Jump-Oriented Programming

It is possible to use gadgets and redirect the program execution flow without the instruction `ret`. This allows to bypass the protections against ROP based on `ret`.

On the x86, an instruction sequence who acts like a `ret` instruction is of this form `pop x; jmp *x;` where x is a general register without `esp`. This sequence is named "Update-Load Branch Sequence" (ULBS). But this sequence is rarer than instruction `ret` and instead of trying to find a turing-complet gadget set that ends by a ULBS, we could reuse one ULBS with sequences instruction that end with an indirect jump instruction. It exists enough instructions ending by an indirect jump to form a Turing-complet gadget set. The jump instruction points to the ULBS and allows to chain the gadgets like a classic ROP. In the following figure, the ULBS is named "trampoline":



ROP without ret

The sequence `pop x; jmp *x;` is not the only ULBS in the x86 architecture, another register instead of `esp` can be used, as long as it fills the same proprieties that `ret` like this sequence `0x4 %eax; jmp *(%eax);`. A defence can not detect all ULBS, so diversifiate ULBS give us more luck to bypass a protection.

4. Instrumentation-based solutions

The instrumentation-based solutions consists mainly to keep a copy of the return address in an area memory named "shadow stack". So, if the return address of a program has been overwritted, the modification could be detected by comparing the value conserved in the shadow stack.

However, this trick cannot stop the others attacks like heap overflows, integer overflows or format string. Furthermore, the control is done only on the epilogue functions and some ROP attacks named "unintended instruction sequences" can bypass the protection. Unintended instruction sequences employ the half of a valid instruction to create a new sequence instruction. Consider the following x86 instruction:

Hexadecimal code	ASM instruction
b8 13 00 00 00	mov $0x13 ,%eax
e9 c3 f8 ff ff	jmp 3aae9

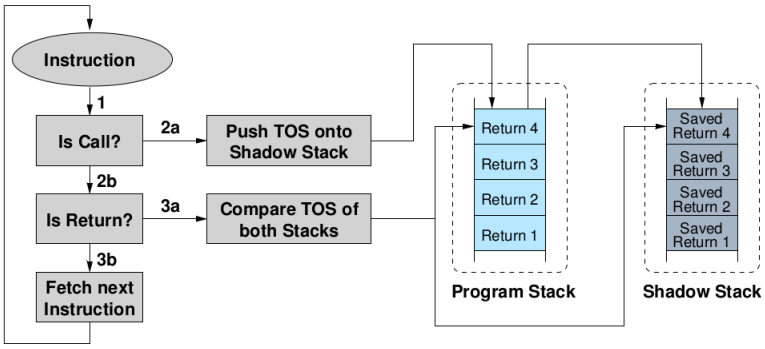
If an attacker points in the middle of the instruction, from the third bytes, we have the following instructions:

Hexadecimal code	ASM instruction
00 00	add %al ,(%eax)
00 e9	add %ch ,%cl
c3	ret

Instrumentation-based solutions generate false positive due to non-management of C++ exceptions, Unix signals, or lazy binding.

ROPDefender is a tool built on the Framework Pin, a dynamic binary instrumentation developed by Intel that uses a just-in-time compilator. It allows to detect ROP attacks using return instruction that bypass solutions based on the control of return only in function epilogues like unintended instruction sequences. The tool does not need additional informations of program.

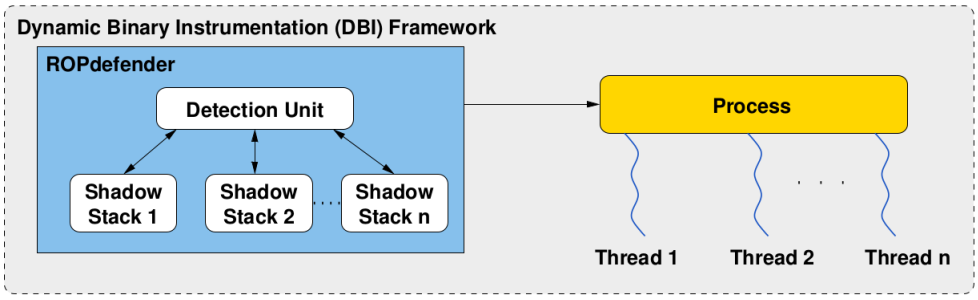
The shadow stack is used except that the return address is saved at each `call` instruction and check at each `ret` instruction.



ROPDefender functionnement
(TOS = Top Of the Stack)

In addition to defending ROP attacks based on return instructions, the tool protects also against all buffer overflows attacks based on the corruption of the return address.

The Pin framework intercepts each instruction before his execution and generates code allowing to observe and debug the program. To avoid false positive due at a multithread program, ROPDefender creates a shadow stack by thread. An invoked function is supposed to return the address of the calling



ROPDefender multithread architecture

function but it exist some exceptions that can provoke false positives. These exceptions can be categorized into three classes.

The first class is setjmp and longjmp instructions that allow, when successive function calls (A calls B that calls C that calls D) to return directly at the

first function (D to A). To avoid false positive in this case, ROPDefender scans the entire shadow stack.

The second class are Unix signals. Unix signals trigger when a particular event comes. When a signal is received, the program invokes the function attached at signal without `call` instruction. The return address of this function is well placed on the stack but not in the shadow stack. Fortunately, the Pin API provides a signal detector that allows at ROPDefender to copy the top of the stack of the program in the shadow stack.

The third class is C++ Exceptions. When an exception is not handle by the function, the return address differs from that pushed by the `call` instruction. The function forwards the exception to the calling function. This procedure is repeated until a function handles the exception, if all functions called do not handle the exception, the default exception handler is called. This is the invoked exception handler that calls destructors of all created objects, this is the "stack unwinding". GNU C++ performs this with _Unwind_Resume and _Unwind_RaiseException functions. These functions call _Unwind_RaiseException_Phase2 that generate the return address and place this at -0xc8(%ebp). ROPDefender copies this address in the shadow stack after _Unwind_RaiseException_Phase2 returns.

Unlike other tools, ROPDefender detects all ROP attacks based on return instructions without informations on the program (source code or debug information), with a minimum false positives due to handle exceptions mentioned above and with better performances (30x at 50x faster). Furthermore, the Pin framework is maintained and continually ameliorated. It is possible to use the tool with a large application case like the Mozilla Firefox browser.

Unfortunately, ROPDefender can not detect ROP attacks without `ret` instruction. Furthermore, though ROP Defender is faster than other instrumentation based solutions and that it can handle most exceptions, performances stay impacted by the control of each instruction.

5. Compiler-based solutions

The compiler-based solutions consist to recompile the code source for prevents the ROP attacks by removing a maximum of gadget. G-Free is a pre-processor for the GNU Assembler that allows to protect against all form

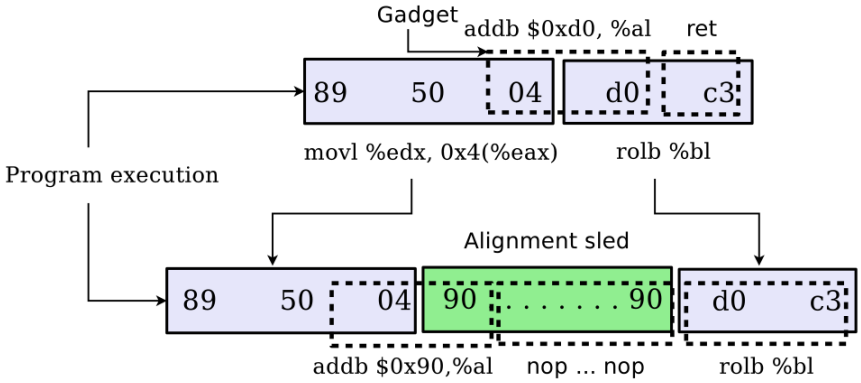
of ROP attacks, including these without `ret` instruction.

The goal of G-Free is to build executables by deleting sequences allowing to chaining gadgets without the user needs to change the source code and to preserve the possibility to optimize the compilation and to implement ASM code.

G-Free eliminates at first the unaligned-free branch instructions inside the executable, so the attacker has no choice to use the `ret` or `jmp*/call*` instruction for chaining his gadgets. And secondly, G-Free protects the aligned free-branch, such as `ret` and `jmp*/call*` instruction, with a combination of techniques that we will see.

In order to protect the aligned branch, three techniques are used:

- Alignment Sleds: The alignment sleds allows to protect against the unintended instructions sequences. That consists by a sequence of instructions that not alter the program execution like the `nop` instruction. This sequence is placed between the critical instructions as in the following figure:



Alignment sleds to prevent executing an unaligned ret (0xc3) instruction

- Return Address Protection: This technical consists in adding in each beginning of functions, that contains a `ret` instruction, a header that encrypts the saved return address loaded in the stack and a footer just before the `ret` instruction for decrypting the saved return address, so preventing the flow redirection through the overwrite of the return.

- Frame Cookies: Against Jump Oriented Programming, the technical looks like the return address protection. A header contains a random cookie with a constant allowing to identify a function is pushed on the stack in each beginning of functions containing `jmp*/call*`. Before each `jmp*/call*` instructions, a validation block is added that decrypts the cookie and compares the constant. At the end of the function, an instruction deletes the cookie on the stack.

To eliminates the unaligned free-branch instructions that consists of unintended free-branch opcodes like `0xc2`, `0xc3`, `0xca`, `0xcb` bytes that can correspond at a `ret` instruction, G-Free uses some code rewriting techniques:

- Register Reallocation: The ModR/M and the SIB bytes can contain a value that corresponds to a `ret` opcode. This is due to the utilisation of some pair of registers like `movl %eax, %ebx` that gives a `0xc3` value for the ModR/M byte. For avoid that, G-Free reallocates the registers when such pairs exist.

- Jump Offset Adjustments: `jmp` and `call` instructions can contain free-branch opcodes in their target address. If the opcode is situated in the least significant byte, adding a `nop` instruction changes the offset and consequently, allows to delete the unintended opcode. If the byte is situated somewhere else, the function or code chunks can be relocate.

- Immediate and Displacement Reconstructions: Some mathematical, logical or comparison operations can contain unintended opcodes like `addl $0xc3, %eax`. We can rebuild the instruction by a equivalent sequence allowing to delete the opcode like follows:
```
addl $0xc1
inc %eax
inc %eax
```

- Inter-Instruction Barriers: `jmp*`/`call*` opcodes appear when the last byte of the instruction is `0xff` and when the first byte of the next instruction contains a suitable opcode. In order to delete the unintended opcode, an useless instruction can be added between the 2 bytes. The `nop` instruction can not be use because it would form a indirect call with `0xff`. We can choose an instruction like this: `movl %eax, %eax`

Because G-Free does not delete free-branch but just protects them, the executable can again contain some gadgets. However, the gadget number re-

maining will be very limited. The pre-compilation has a low cost and increases reasonably the file size.

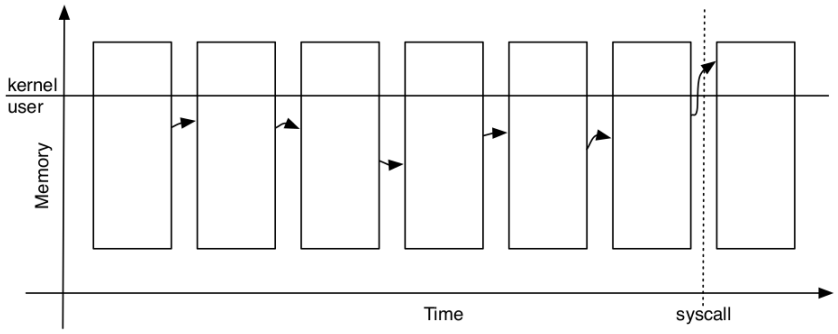
The flaw of G-Free implies that each library need to be pre-compilate with the tools which can generate false positives with library that are not compatible. Furthermore, the compiler-based solutions need to have the source code and are efficient only if all softwares vendors use these compilators. These solutions need more resources and often, the security is sacrificed for the performance.

6. Hardware-facilitated solutions

The hardware-facilitated solutions uses specific hardware features for detect ROP attacks.

kBouncer is a runtime control flow transfers approach. Contrary to others solutions that control all transfers and beget a loss of performance, kBouncer controls only the last branch of a ROP attack, that is points to a call system.

Given that call systems are situated in the kernel memory area, the last step to a ROP attack is to jump since user memory area to the kernel memory area. It is at this step that kBouncer checks if the call system is legitimate or not. If a `ret` instruction points to an instruction that is not preceded by a `call`, a ROP attack is detected.



Steps of ROP attack

For do this, kBouncer needs a hardware features named Last Branch Recording (LBR), a branch trace mechanism added in recent CPUs. LBR allows

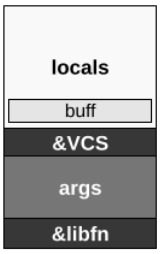
CPU to store the last executed branches in a specific register accessible by a special instruction `rdmsr`. So, the control is transparent, independent of the execution, does not require information of the program, a recompilation or an instrumentation.

However, kBouncer does not detect the JOPs and of course, the systems having not LBR can not beneficial of the protection.

7. Return-to-zero-Protection attack (Ret2ZP)

The common ROP attacks use functions from shared libraries charged by the program. These attacks are named Ret2libc. ARM-processors are invulnerable to these attacks because the functions arguments are passed in the registers instead of the stack. In 2011, Itzhak Avraham presented the Return-to-Zero-Protection (Ret2ZP) attack that allows to exploit the functions of shared libraries on ARM-based systems.

To control values of argument registers r0 to r3 and so execute our functions with the desired arguments, the Ret2ZP attack needs a Vulnerable Code Sequence (VCS) that presents in the system. A VCS allows to copy datas from the stack to argument registers. With a buffer overflow, the function address and arguments are placed on the stack like the Ret2libc attacks and then the VCS address is pushed as it is illustrated the following figure:



Stack contents during Ret2ZP attack

When the program is redirected to the VCS, this one places the arguments in the argument registers r0 to r3 and places the function adress in the `pc` register.

A VCS is a sequence of instructions present in the memory like a gadget but

must respect the following constraints:

- The final instruction must copy the data from the stack in the `pc` register
- The sequence must not contain an instruction who write in the `pc` register excepted the final instruction
- The sequence must copy the data from the stack to the argument registers
- The sequence must not write on the stack

Here is a VCS example:

```
ldm sp, {r0, r1}
add sp, sp, #8
pop {pc}
```

The first line loads argument registers r0 and r1 from the stack through the `sp` register. The last line executes a function return by placing the pop value in the `pc` register.

The researchers Zi-Shun Huang and Ian G. Harris has developed an algorithm that allows to detect VCS:

1. "R" contains registers argument stack-controllable
2. Browse the executable, instruction by instruction "ins"
   - 2.1. Browse all registers argument "r"
     - 2.1.1. If "r" receives a value from the stack then it is added in "R"
     - 2.1.2. If "r" receives a value that does not come from the stack then it is deleted of "R"
   - 2.2. If "ins" writes data on the stack OR if "ins" takes an address that does not come from the stack then "R" is emptied
   - 2.3. If "ins" copies a data from the stack in the register `pc` AND "R" is not empty then

     A VCS is detected and R is emptied

8. Conclusion

The ROP attacks allow to bypass NX protection and ASLR. The `ret` instruction is not the only way for chaining gadgets. Some instructions like `pop x; jmp *x;` have the same properties that `ret` instruction. This variant of ROP attack is named Jump Oriented Attack (JOP) and allows to bypass the protection based on `ret`.

We can categorized solutions against ROP in three approachs:

- Instrumentation-based solutions: control the execution flux of a program and often a shadowstack is used. ROPDefender is one of the best tools developped and can be applied without additionnal information but the instrumentation has a cost and it can not protect against JOP.
- Compiler-based solutions: The executable is recompiled for delete a maximum of gadget. G-Free is one of rare (or the only one) tools that can protect the executable against both ROP and JOP but needs to have the source code and needs that all libraries attached at the program are compatible.
- Hardware-facilitated solutions use features hardware against ROP attack. kBouncer is a solution that had known a great success, the author Vasilis Pappas won the Microsoft BlueHat Prize in 2012. The tool uses the Last Branch Recording (LBR) for detecting ROP attack on runtime with a low cost but cannot protect against JOP.

To finish, we have seen the ret2ZP, an equivalent ret2libc attack for ARM platforms and the necessity to have specific protection solutions for this architecture.

Bibliography

[1] M. Prandini and M. Ramilli. Return-oriented programming. *Security Privacy, IEEE, .vol. 10, no. 6*, pages 84–87, Nov 2012.

[2] S.Checkoway, L.Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without return. *CCS '10*, pages 559–572, 2010.

[3] A.-R Sadeghi L. Davi and M. Winandy. Ropdefender: a detection tool to defend against return-oriented programming attacks. *ASIACCS '11*, pages 40–51, 2011.

[4] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: defeating return-oriented programming through gadget-less binaries. *ACSAC '10*, pages 49–58, 2010.

[5] V. Pappas. kbouncer: Efficient and transparent rop mitigation. *Columbia University*, Apr 2012.

[6] Z.-S Huang and I. G. Harris. Return-oriented vulnerabilities in arm executables. *Proc. IEEE Conf. Technol. Homeland Security*, pages 7–12, Nov 2012.

[7] T. Zheng Z. Huang and J. Liu. A dynamic detective method against rop attack on arm platform. *Software Engineering for Embedded Systems, 2012 2nd International Workshop on*, pages 16–22, Jun 2012.