FH Wedel

Seminar IT Security

Return Oriented Programming

Author:
Julian Wefers

Supervisor:
Prof. Dr. Gerd Beuster

WS 2012/2013

Contents

1 Introduction

In this work, the basic principles and applications of **Return Oriented Programming** as described by *Ryan Roemer*, *Erik Buchanan*, *Hovav Shacham* and *Stefan Savage* will be presented, the historic origins and architecture specific details as well as counter measured will be covered.

1.1 Origins: Code injection

The history of software attacking techniques is long and diverse. Closely related to our topic of Return Oriented Programming are methods of code injection. The idea of code injection is to use buffer overflows (see section 4.1) or similar attack vectors in existing software to insert fragments of machine specific code into the ram of a target computer system and then force the computer to execute this code.

On most computer architectures used today, this is possible because the computer does not distinguish between code and data. Both are series of bits and bytes and it is due to the interpretation of the CPU, that these series are treated as data or executable code. However, vendors of computer hardware spent a lot of effort into closing this point of attack. The most important counter measure is called W$\oplus$X (W xor X), where the areas in the memory are marked with a flag stating whether this area contains contains executable or writable data. An area can never be both. Writable memory segments contain data while executable segments contain program code. To load new code into the memory, the operating system will write the code into a writable memory area and then mark this area as executable, preventing any further writes to this area. Therefore, the operating system remains the only instance to load new code. An attempt to inject code through a software vulnerability would only affect memory areas which are marked as writable. Several names from several vendors exist for this technique, these are DEP (Data execution prevention) on Windows, NX (No eXecute) from Intel, XD (eXecution Disabled) from AMD.

To circumvent this memory protection is the aim of return oriented programming. Since new code can not be injected using the traditional way, the idea is to use and recombine sets or series of existing instructions resulting in new, previously unintended behavior. For a successful attack of this kind, it is necessary to gain access to the stack. To understand how this is done, we will cover some basic concepts about how a modern CPU works on a low level.

1.2 Function calls on ASM level

An important aspect are how function calls work inside the CPU. Therefore, we first need to look at some machine instruction from the x86 instruction set.

The control flow of a CPU is determined by the program counter, a hardware register which holds a pointer to the currently processed machine instruction. After each instruction, the memory management unit increments the program counter to execute the next instruction. Special instructions can now influence the program counter to transfer control to other instruction sequences. These are for instance *jump*, *call*, *loop* and various conditional jump instructions. The *call* instruction is the interesting one for this topic. On the x86 architecture, each process has a stack. When a function is called with the *call <label>* instruction, the CPU will store the current instruction address on top of the stack and then load the address of the called function into the program counter. At the end of each function, a *ret* statement will store the top value from the stack into the program counter and thereby transfer the control flow back to the callee. It is the programmers or the compilers responsibility, that the top value at this moment is the previously stored address.

2 Architectural characteristics

For understanding the ways in which ROP can work, we need to look at some characteristics of today's computer architectures. Three aspects will be covered here, which are Memory alignment, the Harvard architecture and the Von Neumann architecture.

2.1 Memory alignment

Modern CISC architectures have a rich set of machine instructions which vary in length of their opcodes and their parameters. As a consequence, a CISC CPU can start to interpret a single command at an arbitrary memory position (measured in whole bytes). A consequence of this is that instructions may not be interpreted as intended. Take a look at the following opcodes and their representation in x86 assembler:

f7 c7 07 00 00 00	test $0x00000007, %edi
0f 95 45 c3	setnz -61(%ebp)

This sequence tests a register against a bit pattern and sets a byte on the stack if the pattern is not fulfilled. Now if we start the interpretation one byte further, beginning with *c7*, the meaning of the code changes entirely and is even

revealing a *ret* statement at it's end:

c7 07 00 00 00 0f	movl $0x0f000000, (%edi)
95	xchg %ebp, %eax
45	inc %ebp
c3	ret

This conceptual vulnerability makes it possible to use code sequences and instructions that were never intended to exist.

Most RISC architectures have an aligned memory architecture, meaning that all instructions and data have a fixed length or a multiple of a fixed length. The CPU will throw an error if a load instructions tries to access a non-aligned address. Therefore, mis-intended instructions do not exist and and attacker can only use existing and intended code sequences.

2.2 Von Neumann Architecture

Nearly all of todays used computers are based on the Von Neumann architecture. Its main characteristic is a shared memory model, meaning that data and instructions lie together in the same main memory and are accessed through the same interface by the CPU. Whether a data is considered an instruction or ordinary data is decided by the CPU, depending on its actual instruction. For instance, one could inject a byte string into the system which can be executed as a valid instruction sequence. The fact that data can be interpreted as instructions if the CPU is told to, creates a much larger area for the search for ROP gadgets.

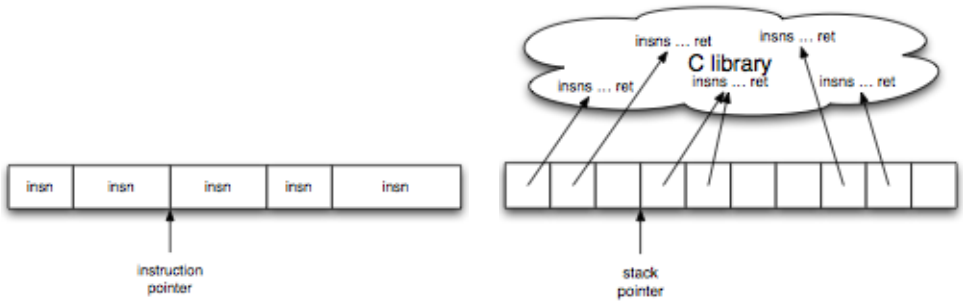
2.3 Harvard Architecture

The Harvard architecture defines, in contrast to the Von Neumann architecture, physically separated memory areas for data and instructions. As a consequence, data and instructions can not be used interchangeable, thus preventing successful code injections. Today, this type of architecture is mostly used in Digital Signal Processors (DSP) and microcontrollers which fulfill very specialized tasks very fast. There are also election machines, which follow the Harvard architecture characteristics.

3 Concepts of ROP

3.1 Idea

To attack systems, which prevent effective code injection by W⊕X or by having physically separated memory areas for data and instructions, we will not try to inject our own code into the system but instead make use of the existing code base that lies inside the memory already. However, it is very unlikely, that we will find a piece of code that does exactly what we intend to do. Instead of searching for exactly one fitting instruction sequence, we search for fragments of existing code that we can chain together in new ways to achieve new behavior. These fragments are called *gadgets* in return oriented programming. The gadgets must end in a *ret* statement, which is basically an unconditional jump to an address lying on top of the stack. By placing the starting addresses of our gadget on the stack, each gadget will transfer control flow to the next gadget by loading it's successor's address into the program counter. As a consequence, we have to inject a sequence of addresses, parameters and immediate values onto the stack that may be necessary during execution.



(a) Ordinary program layout (b) ROP program layout

3.2 From traditional programming to ROP

A traditional program is made up of instructions, it supports the concepts of sequences, alternatives and repetitions. We will see that these concepts are satisfied in ROP and how they are realized.

Sequence On x86, the program counter, called *EIP* (extended instruction pointer) is responsible for executing instructions as a sequence, as it holds the address of the currently executed instruction and will be implicitly incremented

after each instruction and can be changed by *jump*, *call* or *ret* instructions. In ROP, we chain gadgets together by means of *ret* instructions. The addresses for our gadget lie on the stack and are pointed to by the stack pointer (*%esp*, extended stack pointer). *%esp* takes the place of the *EIP*, therefore we need possibilities to manipulate the stack pointer. The auto increment is realized through the *ret* statement, as it increments the stack pointer to point to the next address, that we placed on the stack. In this sense, a ROP program consists of a specific stack layout, containing words that point to ROP instructions (Gadgets) and other values that are used during execution.

For unconditional jumps, we need to find gadgets that change the *%esp* in favorable ways. It is important to understand, that an unconditional branch does not mean to jump to another instruction series, but rather jump to another stack area, where we stored another series of gadget addresses and immediate values. Such a jump may be done, for instance, by using a gadget *pop %esp, ret*, as is can take the address of our next stack segment and store it in the stack pointer. The *ret* will now take it's next gadget address from the new stack segment. As we can see, the concept of sequence is satisfied by using the *%esp* to sequentially execute gadgets.

Alternative Alternatives are realized as *Conditional Jumps* in machine code. They are a very difficult to realize in ROP since it is very unlikely to find sequences that change the *%esp* conditionally (as our %esp is basically our new %eip). So, we are forced to synthesize our conditional branches out of the existing instructions. In x86, a *cmp* statement would set a flag in the flag pseudo register (called *%eflags*). Often, the cmp instruction is not necessary, because many arithmetic instructions set flags as side effects. Conditional jump instructions will then jump according to the condition of a certain flag. However, these jump instructions work on the %eip register and are thus useless for ROP. To conditionally manipulate the %esp, we will load the flag of interest from the %eflags register into a general purpose register. Since we can now deal with a general purpose register, we can for instance conditionally set a relative distance and then add this value to the %esp. Another, not always possible strategy would be to find gadgets, that encapsulate our desired branch behaviour in ordinary fashion. For this, we write the values on which our branch decision relies into general purpose registers and then execute gadgets, which compare these values and perform some sort of branching.

Repetition Repetitions can be expressed as conditional jumps to an earlier point in the code (earlier in the sense of execution time, not in memory position). In a repetition, we want to repeat certain sequences of core more than once.

However, since we want to use a gadget sequence more than once, we must make sure that the stack stays intact, even when we consume a gadget address from the stack.

Immediate Values In ordinary programming, immediate values like constants are often stored in a central place or may even be hard coded into the program. As it is very unlikely to find gadgets that contain the exact desired values we will use a different approach to deal with immediate values. As previously mentioned, our prepared stack may not only contain gadget addresses but also immediate values. These immediate values can be accessed through operations like *pop*, that loads a value from the stack into a register of choice.

traditional	ROP
mov 0x123 %eax;	pop %eax;

As you can see, instead of just loading a value into a register, we take a value from the stack, which we placed there before.

Variables This can be done just as in ordinary programming. Variables can be stored for example on the heap and can be made accessible by references in our crafted stack or by using existing variable locations in the code base we are using. If we want to use these pre-existing locations, it may be necessary to make use of load/store gadgets, to prepare these locations before accessing them.

3.3 Search for Gadgets

A ROP program consists of a series of these previously introduced gadgets. Since we have to rely on the existing code base on the target system, we have to search for these gadgets. We know, that each gadget must end in a *ret* instruction (on x86) and we can search with a granularity of one byte, leveraging unintended instructions as well. In any given library like libc plenty of these sequences can be found. To ensure correct execution of a gadget, the following condition must be fulfilled: The *%esp* must point to the gadget and the CPU must execute a *ret* statement. The *ret* will instruct the CPU to transfer the control flow to the address pointed to by the *%esp*. The gadget is then executed.

Since it is necessary to know the target system, one can build a library of existing code containing information about all possibly useful instruction sequences. This search must contain all immutable memory areas of the system, which are most often system libraries which will always be found at certain memory areas. The search can be done by seeking the memory for return instructions and then

backtrack to index all possible instruction sequences. This is a possible data structure for such a catalog:

```
data Codebase = Node Address [(Instruction, Codebase)]
```

The structure describes an n-ary tree, where the root is always a ret statement. Each node contains the address of the beginning of an instruction sequence and a list of existing prefixes for this instruction series, each pointing to the starting address of this prefix. While we will almost definitely find more than one address for certain instruction sequences, it is sufficient to store only one of them. When building up the index, we have to backtrack from all found instructions in each step to accurately fill it, but only store one address for each possible sequence. It also means, that the addresses, following from a leaf to the root, do not necessarily represent a connected instruction sequence. When looking for a specific code sequence, one would start at the root and descend in the tree, following the intended instruction series backwards. If only a part of the desired code can be found, save the address from the last node in the tree and start from the beginning. This means, that we split out code into two or more gadgets. For each gadget, we will later save its address at our prepared stack.

For an architecture without memory alignment, we have to slightly alter our indexing, since we can take unintended instructions into account. So, beginning at a ret instruction, we have to go backward bytewise and check for each byte or byte sequence, whether it is a valid instruction. This leads to the consequence, that we can have more than one path through the tree following the same byte pattern in the code base, but it rewards us with more possible code sequences.

In some cases, we will be forced to express a simple operation, that we can not find in out library in an obvious matter, through other, possibly more operations. When writing a ROP program, or compiling to a ROP program, one must be able to identify other constructs. For instance, if we want a value in register %eax, but can not find a gadget *pop %eax; ret* (for the sake of the example), we could instead use an instruction series like *pop %ebx; mov %ebx, %eax; ret* to achieve the same result, provided we don't need a value in %ebx at that time.

4 Attack vectors

To start a ROP program, it is necessary to disturb a running program to start the execution of our ROP program instead. Several possibilities exist.

4.1 Buffer Overflows

A common attack vector for all sorts of exploitations are buffer overflows. This is a situation, when a write operation writes beyond the intended bound of a memory area. For instance, every array has a given size. In low-level languages, boundary checks are in the responsibility of the programmer. When we start to fill up the array with values, but do not check if we are still within bounds, we can write beyond the array's allocated memory. This is called a buffer overflow. In the example below, the buffer is an array, which we overflow by writing past its bounds. Below, such a case is demonstrated by reading characters from stdin and writing them into a buffer of size 6. However, the *scanf* function may read an arbitrary large number of bytes and will store them starting at the beginning of the buffer. Enter more than 5 characters (because a string needs a termination byte) and *scanf* will write past the upper boundary of buffer and into the stack on which the array lies.

```
int main( int argc, char * args[] )
{
  char buffer[6]; /* 5 chars + \0 byte */
  scanf( "%s", buffer );
  return 0;
}
```

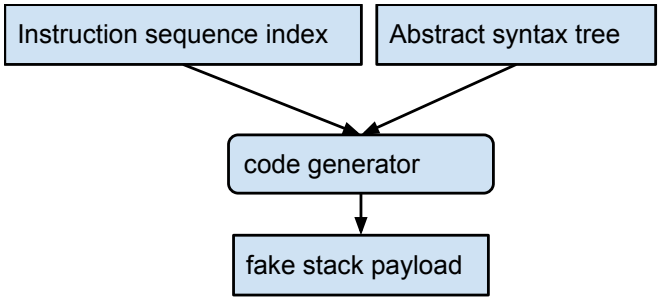
After finding such a vulnerability, an attacker might use this to craft a string, which holds 6 arbitrary characters and then continues with machine code patterns for a target architecture. The effect would be that past the upper boundary of the buffer lies data which can be interpreted and executed as correct machine code.

Buffer overflows commonly exist in software written in language that offer a low level of abstraction. While these allow for greater speed of written software, it is the programmer's responsibility to ensure correct handling of input data. This often leads to flawed programs and makes buffer overflows a commonly used vulnerability to attack software of all kinds.

When attempting to inject a fake ROP stack through a buffer overflow, the goal is to find and exploit a buffer overflow to write our rop payload onto the overflown stack, containing the address of the first gadget on top of the stack, so that when the function, where the buffer overflow exists tries to return, the first gadget of our rop program is executed instead. Note, that a function may have several more local variables that lie beneath the buffer that we overflow. In this case, we have to prefix our payload with the exact amount of dummy values to place our first gadget address in the right position for the ret statement of the attacked program.

5 ROP compiler

By now, it should be clear that crafting a non-trivial ROP program by hand is not very favorable. But since we showed how to create and populate an index of existing code structures, we can actually automate the creation of a ROP program and write our code in familiar high level languages like c. A compiler consists of Phases which are lexing, parsing, generating code, and possibly a dedicated optimization phase. Parsing and lexing can be done just as usual, the interesting part is the code generation, as it can not simply construct instructions but has to map the desired result to existing instruction sequences.



The output of the code generation phase is not an executable binary, but a fake stack payload that can be injected onto the actual stack using exploit techniques. Since the stack must reference existing code sequences, the construction of gadgets is the main task for a code generator.

With the data structure for indexing instruction sequences given, the next step is to search for sequences within that structure, that resemble our commands written in the high level language. If an exact sequence can not be found, the compiler must attempt to transform the problem equivalently to find a gadget sequence, that produces the same result while avoiding unwanted side effects.

6 Counter Measures

Some approaches exist to prevent attacks that rely on exploiting an existing code base. The most important of them are:

6.1 Address Space Layout Randomization

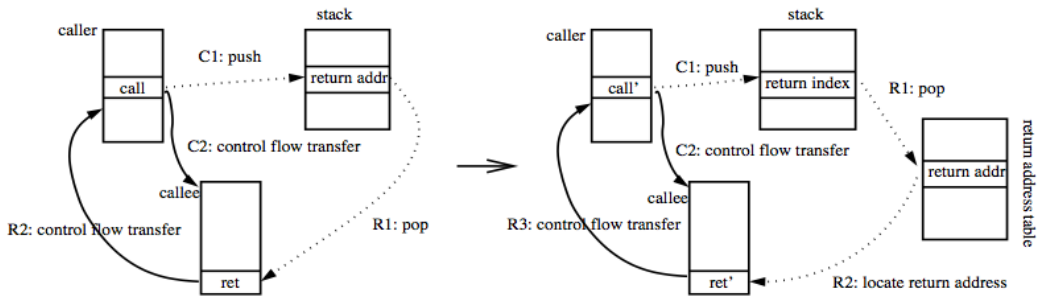
ASLR is a technique that randomizes the positions of key data areas such as system libraries, stacks and data segments for processes. It is implemented at

the OS level and is ideally influencing the OS startup as well, also randomizing the position of essential code like the kernel and it's libraries. The randomization makes it very difficult for an attacker to guess the position of required code sequences in advance. The security gain of ASLR obviously depends on the size of the search space and the entropy of unused memory areas. A higher entropy can be created by filling empty spaces of memory with random bytes, making it harder for an attacker to properly locate useful code sequences since a grid search for non-empty areas can not decrease the search space. A higher search space can easily be achieved by increasing the amount of virtual memory that is used by the OS.

Today, all major OS vendors have implemented ASLR into their operating systems on kernel level. However, a recent exploit targeting Adobes PDF Reader, presented in section 7.2 used a previous stage of malicious embedded code to circumvent ASLR by constructing the gadgets at runtime.

6.2 Return-less kernel

Return-less kernels are a compiler based approach, targeting to introduce an indirection to return addresses. In normal control flow, we usually deal with *call* - *ret* pairs. A *call* will push the return address to the stack and then transfer the control flow to a given address. The *ret* statement will load the address to return to into the program counter and increment the stack pointer. It essentially performs *pop %eip*. The indirection to create return-less kernels consists of having a separate table for all used return addresses, and instead of using the call instruction to push an immediate address onto the stack, an index is pushed to the stack, pointing to an address in the table. Likewise, we will not use a return statement to transfer control flow back to the caller, but instead use the pushed index to retrieve a concrete address from the table and then jump to this address.



(a) Traditional control−flow transfer

(b) The new control−flow transfer based on return indirection

This disables an attacker from choosing his own return address, since the table can be initialized and populated offline, therefore marking it as read-only, like all kernel data. However, this approach does not prevent attacks like return-into-libc, which rely on complete and legitimate functions. This attack can still be driven by overwriting the table index. But since ROP is a generalization of previous attacks in the sense of return-into-libc, a Return-less-kernel degeneralizes and limit the possibilities of exploiting existing code. One notably effect is the restriction of branching that ROP offers and which makes up for Turing complete computing.

6.3 Runtime-based detection methods

Several approaches exist to detect (not prevent) the usage of ROP during runtime, thus giving the user the possibility to pull the plug for further investigation. Two of them are introduced here:

Detecting frequent returns One approach called *Dynamic Integrity Measurement and Attestation* (DynIMA) proposes the use of runtime checks for several situations: The first is to mark untrusted data and terminate a process execution if that data is misused, e.g. as a pointer. Second, the instruction count between two ret execution is measured and an alert is raised if that count goes higher than 5. This relies on the observation, that most rop gadgets contain only 2–5 instructions before calling the next ret. [5]

Stack shadow copy This approach tries to keep a shadow copy of the stack somewhere in the memory and runs comparisons from time to time against the real stack. However, it is difficult to maintain a synced stack and thus an algorithm making the comparison needs a trigger implementation to detect, when the stack differences are big enough to raise an alarm (possibly due to a change of the stack pointer to an entirely different memory section).

7 Existing Applications of ROP and further development

Although ROP is a very young technique, first presented in 2007, ROP has found a number of noticeable application in real-world attacks and as implementations in common exploit kits. Some examples will be presented here. In addition, current research activities will be briefly introduced.

7.1 Manipulating a voting machine

In this research case, which was not a serious real-world attack, a group of students, mostly from UC San Diego demonstrated a successful manipulation of a Sequoia AVC Advantage voting machine [6]. This machine delivers some serious security features: The complete software is stored in a read-only memory and the Z80 processor strictly refuses to execute code that was fetched from the RAM by means of hardware. This makes the Sequoia AVC, which design originates in the late 80's a Harvard architecture. As external input, a memory cartridge is used.

The team reverse-engineered the machine and developed a simulator to develop and test their exploit, using the real hardware only at the end to validate their results. An interesting point is, that the Z80 processor has a very dense and variable-length instruction set, resulting in absolutely no invalid instructions. As a result, the exploit could make use of a lot of unintended instructions.

A stack buffer overflow was found in the cartridge processing, allowing the team to craft a physically manipulated cartridge for which the flawed procedure can be activated. The procedure loads several files, one of which is fixed-size, but has several dozen unused bytes in it. At the first stage, the stack pointer is modified to induce a rop jump to a memory section under control by the attacker, namely the position of where the file lies in the memory. The attacker uses the unused bytes to include the necessary gadgets to the attack. In a second step, another file from the cartridge, containing the biggest part of the gadgets is then loaded into the memory and the stack pointer will again be redirected to that file, giving the attacker full control by the means of the injected gadgets.

After the control flow is taken, the machine can be forced to swap and steal votes.

As the team had full access to the target system, this attack might not seem to be relevant for the real world, but in countries, where the current government has an interest to manipulate election outcomes, attackers may easily get full access to actually used voting machines, thus making this case an important study about the security of current systems.

7.2 Adobe PDF Sandbox Exploit

This exploit, as described in an article from Xiao Chen (McAfee) from February 2013 [7], is a fully functional zero-day exploit targeting Adobe's PDF Reader and uses techniques such as a complete ROP payload and highly JavaScript code to escape the Readers Sandbox. It comes as a prepared PDF file, which has embedded some obfuscated Javascript code to manipulate Adobes XML Forms Architecture (XFA), an encoded XFA object and two binary streams which are

believed to be two encrypted DLL's. The attack consists according to Chen of two steps. The first one uses Javascript code to determine an address to one of the readers base modules, to determine the version of the readers version, amongst other things. At this stage, the Javascript code will construct the rop payload based on the leaked information. Since the ROP payload is constructed at runtime, the exploit can actually circumvent ASLR, as the target addresses are retrieved after the target has been initialized. The two ROP parts used in this exploit are used to decrypt the embedded DLL files and drop them to the file system. Malware scanner fail to detect this, because all used code addresses lie inside of Adobes Reader, which is a legitimately running process. Note, that the attacker performed a decryption of files using only ROP code, showing the possibilities of ROP.

7.3 ROP without returns

It this latest work from a team around Schacham, they introduce a new approach to ROP [4], omitting actual ret statements and instead rely on constructs, that achieve the same effect as a ret statement. Such a construct on x86 can be for instance *pop x; jmp *x* where we load an address from the stack into a general purpose register and then jump to this address. The most convenient statement would be *pop %eip* which has practically the same effect as a *ret* statement. Observations have shown, that these sequences do not show up with enough frequency to find a turing complete set of gdgets that end in these sequences. So instead of searching for gadgets that end in such sequences, it is sufficient to find one of these constructs and reuse it. Therefore, our reusable ret replacement will be called a trampoline as we use it just to transfer the control flow to the next gadget. The reuse is achieved by searching for instruction sequences that end in a so-called indirect jump to such a location, e.g. *jump x* where x is a general purpose register. The difference to the trampoline is, that we will not attempt to jump to an address from the stack (which would be our gadget list) but instead store the well-known address of our trampoline in a general purpose register through other means. Sequences ending in indirect jumps appear with a sufficient frequency to use them to construct a Turing complete gadget set. To enhance the concept, it is possible and depending on the target code base reasonable to introduce more than one layer of indirecten, e.g. when the address of the trampoline can not be constructed into a general purpose register but only another sequence, itself resembling an indirect jump to the trampoline. This new concepts renders the use of return-less kernels obsolete, when properly used and therefore shows, that no security concept can be final.

8 Conclusion

We have seen that Return oriented Programming is a decent and powerful, yet complex exploiting technique, which can be seen as a legitimate successor to code injection and return-into-libc attacks. One of it's main advantages is, that it delivers the possibility of arbitrary, turing complete computations without having to construct a multi-staged exploit with adventurous. This flexibility did not came overnight but was the result of years of ongoing research and evolving attack methods. Firstly presented in 2007, ROP has already found it's way into the hand of malicious code writers that injected rop-based code into zero-day-exploits of avoid detection from malware scanners and similar approaches. However, counter measures exist and are actively being developed but new research always invents new ways of circumvent these counter measures, showing again that IT security will always be an arms race.

The development of a functional rop attack requires however, fundamental knowledge about computer architectures at a low level and is very expensive. But with the upcoming of ROP exploit kits and compiler, even medium talented cracker are now able to construct functional ROP exploits, making this technique a serious threat for computer systems of all kinds.

References

[1] *Return-Oriented Programming: Systems, Languages, and Applications*; Ryan Roemer, Erik Buchanan, Hovav Shacham, Stefan Savage; ACM Trans. Info. & System Security; http://cseweb.ucsd.edu/~hovav/dist/rop.pdf

[2] *Defeating Return-Oriented Rootkits With "Return-less" Kernels*; Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, Sina Bahram; Proceedings of the 5th European conference on Computer systems, Pages 195-208;

[3] *On the Effectiveness of Address-Space Randomization*; Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, Dan Boneh; Proceedings of the 11th ACM conference on Computer and communications security, Pages 298-307; http://www.cs.columbia.edu/~locasto/projects/candidacy/papers/shacham2004ccs.pdf

[4] *Return-Oriented Programming without Returns*; Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, Marcel Winandy; Proceedings of CCS 2010; http://cseweb.ucsd.edu/~hovav/dist/noret-ccs.pdf

[5] *Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks*; L. Davi, A.-R. Sadeghi, and M. Winandy; Proceedings of the 2009 ACM workshop on Scalable trusted computing; 2009, pages 49–54. ACM Press, Nov. 2009.

[6] *Can DREs Provide Long-Lasting Security? – The Case of Return-Oriented Programming and the AVC Advantage*; Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, Hovav Shacham; Proceedings of EVT/WOTE 2009; http://static.usenix.org/event/evtwote09/tech/full_papers/checkoway.pdf

[7] *Analyzing the First ROP-Only, Sandbox-Escaping PDF Exploit*; Xiao Chen; http://blogs.mcafee.com/mcafee-labs/analyzing-the-first-rop-only-sandbox-escaping-pdf-exploit