Exploitation vs. Prevention: The Ongoing Saga of Software Vulnerabilities

László Erdődi, Audun Jøsang

University of Oslo, Gaustadalléen 23 b, 0371 Oslo, Norway
laszloe@ifi.uio.no, josang@ifi.uio.no

Abstract: Online IT systems are frequently exposed to cyber-attacks. An Exploit is an advanced attack tool that takes advantage of some software vulnerability to attack and cause harm to IT infrastructures. Developers and manufacturers of operating systems and hardware put huge effort into the prevention of vulnerability exploitation (e.g. Data Execution Prevention, Control Flow Integrity, etc.). However, the number and severity of attacks show that new exploit methods are continuously being invented despite the increasingly sophisticated protection methods. The present article summarizes the current, known and most relevant software vulnerability exploitation methods, as well as, the possible methods used to protect against these exploits. Moreover, the effectiveness of both the exploitation and prevention methods (as seen from both the attacker's and the defender's sides) is analyzed to find a possible future direction, to eliminate exploit attacks against an IT infrastructure.

Keywords: vulnerability; exploitation; protection; control-flow

1 Introduction

Software coding errors can become vulnerabilities that can allow malicious exploits to take control over computer systems. Using deliberately malformed input data attackers can cause unintended or unanticipated behaviors in a software package that contains a particular type of vulnerability. Depending on the type of vulnerability an exploit can be a sequence of commands, a chunk of data or a piece of software to cause malicious code execution for the sake of the attackers. Exploits can be categorized according to their capability (e.g. remote code execution, DOS), the platform they can be applied to (e.g. Windows, Linux, IoS, etc.) and also according to the way of execution (local, remote). Some websites allow the public to register known exploits, such as, the exploit database [1], where users can submit ready-to-use exploits. Exploitalert [2] is another website that reports exploits with detailed data found on the Internet. Another exploit collection is the Metasploit framework [3] which contains several exploits in a

unified form which makes the exploitation very easy and automatic for the attackers.

Figure 1 shows the number of the available exploits over the years, according to the Exploit database [1]. Even if this figure and the available sources do not contain all the existing exploits, it is nevertheless, interesting to observe the trend. The number of new exploits was on the top in December 2009 when nearly 600 new exploits were added during one month. The Data Execution Prevention (DEP) [4] and the Address Space Layout Randomization (ASLR) [5] became basic feature of operating systems around that time, which can explain the significant decrease in the number of new exploits after 2009. Another reason for the decrease can be the appearance of the dark web.

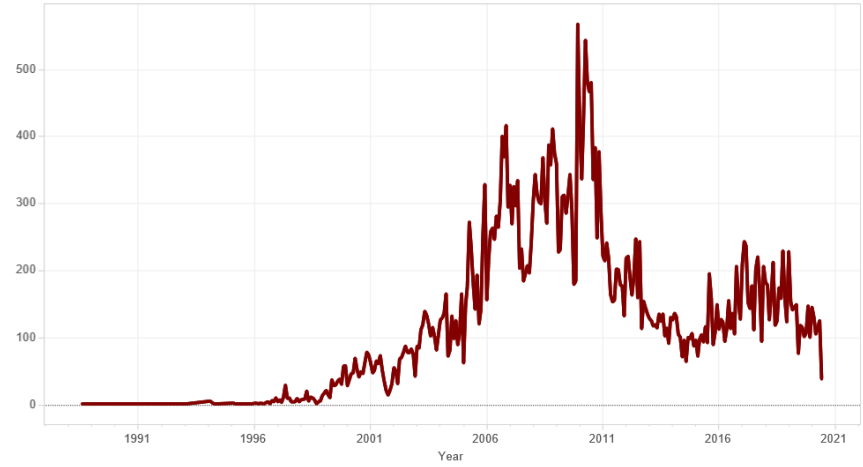

Figure 1

Number of recorded new exploits per month in the exploit database [1]

An exploit is usually able to take advantage of one particular vulnerability in a particular piece of software, but there are some exceptions. A general exploit can affect multiple platforms as it customizes itself for the actual version of the software. Some exploits use two or more different vulnerabilities at the same time to achieve their goals [6]. For a modern web browser exploitation, sometimes three different vulnerabilities are necessary: one for obtaining the ASLR randomization offset, one for exploiting the vulnerability and a third one to break out from sandboxing.

From a vulnerability point of view, two major categories can be created according to our categorization: The configuration error based and the software error-based exploits. The exploit that takes advantage of a configuration error can use e.g. default passwords, access hidden content or bypass protections by misusing the system. In all of these cases the vulnerability is connected to inappropriate configuration. In this paper we focus on the other case when the configuration is

correct, but the software code contains vulnerability. Since we use different software layers that are based on each other the bug can be on different levels too. The level of the bug significantly determines the difficulties of the detection and protection possibilities. For example, a Content Management System (CMS) uses a kind of server side scripting code which is executed by the webserver software of the operating system. The web server software uses the operating system API which is based on the kernel level code of the operating system. So a bug in a php code, on the CMS level, has different effect than a bug in a kernel driver. Figure 2 shows the different layers.

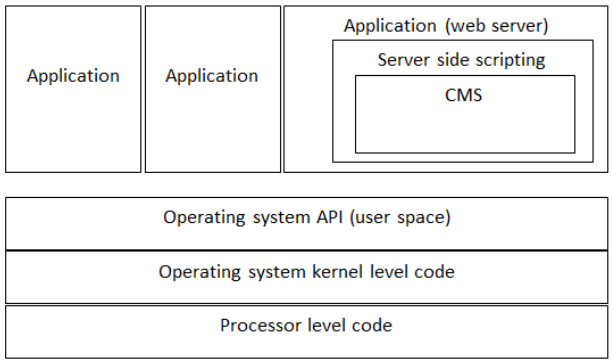

Figure 2
Software code levels

If the vulnerability is e.g. in a kernel driver, then the exploit has the system right to execute the malicious code. In the user space the exploits have the same right as the application that contains the vulnerability. In these cases, e.g. a crafted PDF file is the exploit itself that is opened by the PDF reader (application). If the application provides services, then the attack surface will be increased. In the case of a web server application the vulnerability can be inside the application code or in the high level server side code (e.g. php based SQL injection). In other cases, the Content Management System (CMS) contains the vulnerable server side code (e.g. Drupal SQL injection [7]). Exploits can be created in all of these cases, but obviously the form of the exploit is totally different for a kernel driver bug and for a Drupal SQL injection.

The CVE database [8] contains the distribution of different vulnerabilities. It contains a huge amount of webserver-side coding vulnerabilities but the number of lower level coding vulnerabilities like memory corruption is also significant. This paper focuses on the lower level type of vulnerabilities, where the exploitation is carried out directly within the virtual memory.

We can also categorize the exploits according to the vulnerability exposure date. If the vulnerability was previously unknown, then the exploit would be called a zero day (0day) exploit. In other cases, the vulnerability is known but the exploit is still

actual since the vulnerability is not patched everywhere or cannot be patched. In this case it can be referred as 1st day exploit.

Protecting the system against a first day exploit is usually not a real challenge, because the manufacturer has to provide a patch to remove the security gap after the vulnerability disclosure. The main focus of the exploit prevention is to protect the system against the 0day exploits, when concrete attack signatures cannot be used. This is possible by providing a secure execution environment which prevents the exploitation of an unknown vulnerability of the software. Several exploitation and attacking techniques exist and the main focus is to stop the exploitation without significant resource usage overhead. Since hardware based techniques hardly slow down the normal execution speed they are more preferable. In Chapter 2 different exploitation and protection techniques are summarized, while Chapter 3 focuses on future potential exploitation techniques and their analyses.

2 The Evolution of Software Vulnerability Exploitation and Protection

2.1 Early Exploitations

In the early years of software vulnerability exploitation, the aim was to find some coding error types that could lead to compromises, such as, arbitrary code execution. In this context there is no specific protection against vulnerability exploitation; everything is based on code correctness. The operating system focuses on the fast and efficient code execution within the virtual memory without any protection that considers coding errors. The program code and the shared libraries are loaded into the virtual memory to a code segment of the virtual address space having the operating system API. Each thread of the application has its own stack segment that consists of the method call stack frames. The whole process has some common heaps, where the dynamically allocated objects are stored. Each object has a virtual method table that contains the actual addresses of the virtual methods during runtime. For the sake of the effective and fast memory allocation and free in runtime, every heap is organized as series of linked list chunks with different sizes. A simplified figure of the virtual address space is presented in Figure 3.

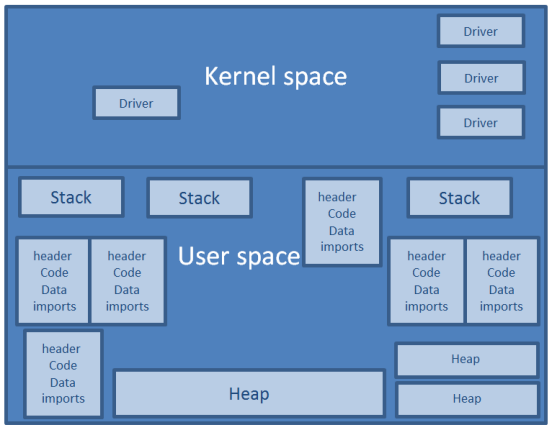

Figure 3
Virtual address space layout

In the early years of exploitation, the security of a software was only provided by the coding. If the code had no vulnerabilities, then the software would not be compromised. Unfortunately, this not the usual case, and with a single coding error, the attacker can force the software to execute malicious code. This malicious code execution is possible using several well-known techniques, such as, the stack overflow [9] the heap overflow [10], the format string vulnerability [11] or the use-after-free bug [12].

In the case of stack overflow [9] a local variable of a method (e.g. a string or an array) is overwritten inside the stack frame. Since the stack frame contains the method return pointer too, the attacker can redirect the execution to an arbitrary place by providing a new return pointer inside the local variable. By placing the attack payload in the corrupted local variable on the stack, the attacker can redirect the execution to the stack itself and the malicious payload is executed there.

In the case of heap overflow [10] the overwritten variable is in the heap. By overrunning a heap chunk the attacker will be able to modify the linked list pointers of the current heap. During the process of merging the freed heap chunks the chunk pointers are used for writing data. With an appropriate pointer modification, the attacker can write arbitrary data to an arbitrary place when the heap is freed. This is the way how the execution is redirected to the code where the malicious content is previously placed.

In the case of format string vulnerability [11] the attacker provides a series of formatting characters of which no data belong to for a *printf* type of functions. Choosing the formatting parameters appropriately, the attacker can write almost arbitrary data to an arbitrary place. By overwriting sensitive data in the virtual memory such as the stack method return pointer or a virtual address table pointer

the execution is redirected to the attacker controlled place where the malicious payload is executed.

The use-after-free exploitation technique [12], is based on the modification of an object virtual method table pointer. If the vulnerability consists of an object that can be used after being freed then the attacker can try to allocate a fake object to the same place in the virtual memory where the original object was to redirect the execution. To achieve this, the attacker has to allocate multiple fake objects, with fake virtual method tables in the heap, that are pointing to the malicious code, that has already been placed in advance (heap spraying). When a virtual method of a freed vulnerable object is called, then the malicious code is executed.

It is easy to draw the conclusion from these early exploitations, that software security cannot be based only on the code correctness; additional protections are also necessary to avoid software bug exploitation.

2.2 Early Protections

The early solutions focused on the protection of the critical data in the virtual memory. For example, the stack frame return pointer overwriting, is aimed to be protected by the stack cookie [13]. As the stack cookie is placed between the method local variables and the method return pointer, any modification outside the real memory range of the local variables results in the modification of the stack cookie too. Therefore, the stack cookie modification indicates the stack frame corruption for the operating system. If stack cookie is placed in each stack frame, then this protection will be good enough to filter the stack frame corruption. However, it comes with a significant speed performance penalty.

The heap chunk header exploitation is prevented by the secure heap chunk unlink process [14] that validates the chunk header pointers before it is merged with another chunk. The secure structured exception handling [15] is another special defense that was introduced early against the exploitation of the exception handling vulnerabilities. This protection validates the exception handler pointer before it is executed.

In addition, several more robust protections appeared in the middle of the 2000s. These protections such as the Data Execution Prevention [4] and the Address Space Layout Randomization [5], aim to make software exploitation more complicated in general. Data execution prevention enforces memory page rights for the different types of segments in the virtual address space. Reading, writing or executing the page data are all different types of operations and DEP ensures that a memory page cannot be written and executed at the same time. DEP stopped several previously mentioned exploitation methods such as the stack overflow, since the payload can be written to a writable memory place but it cannot be executed due to the stack DEP protection.

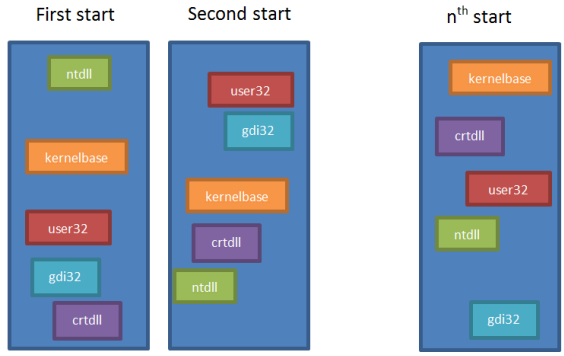

Figure 4
Address Space Layout Randomization in Windows [4]

Address Space Layout Randomization [5] is about to prevent the reuse of the already existing code parts in the virtual memory for malicious purposes. If the locations of the different segments in the virtual memory are randomized every time when the program is launched (Figure 4) then the attacker cannot rely on the known memory addresses of the shared libraries. It is also important to provide sufficient entropy for the randomization to prevent code reuse exploitations with guessing the ALSR offsets.

2.3 Advanced Exploitations

With the introduction of Data Execution Prevention [4], exploit writers could no longer place their own code to be executed. Attackers had to apply new techniques and the main idea became to execute the already existing code parts in the virtual memory that have the right to be executed, and that is the code reuse.

The first applied technique was the *return to libc* [16] type of exploitations where the corrupted method is redirected to an operating system API by placing its address as a return pointer in the corrupted stack frame. However, this technique is can execute only one operating system method but, selecting the right method, such as, the *WinExec* or *Execve*, with the right parameters can be sufficient.

A significant break-through for the code reuse was the invention of the Return Oriented Programming (ROP) [17]. This technique divides the desired payload into small code parts (gadgets) and searches for same code parts in the code libraries in the virtual address space. Since the gadgets are part of the virtual memory there is no need for own code to be placed, the payload is only a series of the gadget addresses and their parameters. Each gadget contains some assembly instructions with a *ret* type instruction at the end. When the corrupted method exits, the execution will be directed to the first gadget by its address. Because of the *ret* instruction at the end of the gadgets, the execution is directed to the next

gadget every time by taking the next address on the corrupted stack frame by the *ret* instruction. ROP is proven to be Turing complete, the only limitation is the gadget catalog provided by the virtual address space. According to our current experiences there is no practical limitation, the attacker can almost always find enough gadgets in the virtual address space to turn off the DEP and continue the payload execution in the traditional way.

Jump Oriented Programming (JOP) [18] is a generalization of ROP and also capable of bypassing the DEP protection in a very sophisticated way. Similarly, to ROP, JOP executes the payload step by step by using small code parts called the functional gadgets. Each functional gadget has an indirect jump instruction at the end to redirect the instruction pointer to a special code part called the dispatcher gadget. The functional gadget addresses are stored in the dispatcher table that has to be placed in the virtual memory before the exploitation.

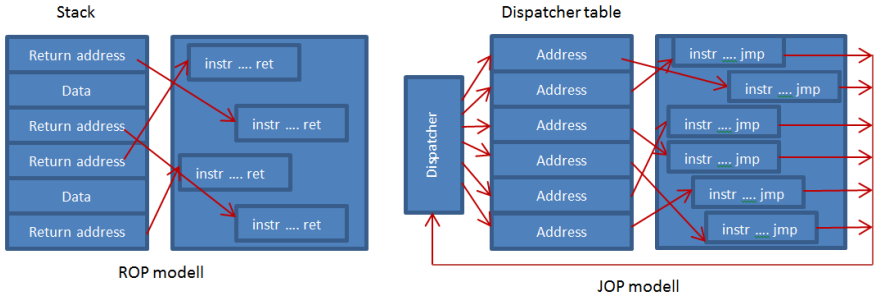

Figure 5
Return Oriented and Jump Oriented Programming [17] [18]

The dispatcher gadget maintains a register which always points to the next functional gadget in the dispatcher table to be executed. Instead of relying on the stack and the *ret* type instructions, JOP realizes its own stack like structure by the dispatcher table and the concatenation of the gadgets are ensured by the indirect jump instructions of the functional gadgets and the indirect call instruction of the dispatcher gadget.

There exist some other forms of scattered code reuse technique and these are under research such as the Sigreturn Oriented Programming (SROP) [19] or the Call Proceeded Return Oriented Programming (CPROP) [20]. SROP is based on the kernel context switching that saves the current execution context in a frame on the stack. The saved execution context contains the saved registers, as well as, the flags. In the case of stack overflow the instruction pointer is overwritten in the saved execution context. This is how the execution is redirected when the OS gets back the register values from the stack to resume the previous context. Contrary to ROP, SROP exploits are usually portable across different binaries and can also bypass ASLR in some cases.

Call Proceeded Return Oriented Programming applies whole functions as a gadget in order to bypass the control flow protections. With this approach every *ret* like instruction is legitimate during the payload execution and cannot be discovered with method return address validations.

Even if bypassing DEP is possible with the listed techniques, it is important to state that the gadget addresses should be known in order to apply these techniques. With ASLR this condition is not satisfied so attackers have to also consider ASLR bypassing, which is always a challenge. In some cases, ASLR can be bypassed by simple guessing the randomization offset [21] or by taking advantage of another vulnerability that leaks the randomization offset [6]. Special techniques to bypass ASLR and DEP together already exist: The Blind Return Oriented Programming (BROP) [22] and Just in Time Return Oriented Programming (JIT-ROP) [23]. BROP maps the virtual address gadgets by systematic guessing, while JIT-ROP does a just in time payload customization relying on an ASLR offset leak.

2.4 Current Exploitations

Secure software development is a fundamental question and several protections exist. Even compilers, operating systems and hardware manufactures try to mitigate software exploitation as much as possible, several exploits are still successful. Analyzing the exploits found in the wild, published by researchers and white hat hackers, it is clear that attackers have to consider the DEP and the ASLR together as a basic elements of the modern operating systems nowadays. Some browser exploits appeared at the end of 2016 and the most popular exploitation method was the Just in time Return Oriented Programming. A Firefox/Tor exploit (CVE-2016-9079) is revealed [24] at the end of 2016 that maps the WindowsPE structure in runtime to find appropriate ROP gadgets. The ROP code turns off the DEP with the *kernel32.VirtualAlloc* method then the rest of the payload is executed in the conventional way. Another DEP and ASLR bypassing exploit is related to the chakra JavaScript [6]. This exploit uses two different vulnerabilities. CVE 2016-7200 is used for the ASLR bypass, the *mshtml.dll* randomization offset is obtained with that bug, while CVE 2016-7201 is used to execute a short ROP code to turn off the DEP. Both cases belong to the Just in Time Return Oriented Programming category. ROP based exploits are used everywhere e.g. against network devices too. A vulnerability (CVE 2017-3881) [25] in the Cisco Cluster Management Protocol (CMP) processing code in Cisco Software could allow an unauthenticated, remote attacker to execute code with elevated privileges.

Based on the currently available software exploits, it is obvious that the main technique is still Return Oriented Programming. DEP and ASLR in combination were thought to provide very strong protection, but the current examples show that they can be bypassed routinely in several cases. The next step from the protection point of view is to disable ROP, where a possible approach is to enforce the right

control flow during the code execution. In Section 3 several control flow bypassing techniques will be analyzed.

2.5 Enhanced Protections

Because the software vulnerability exploitation is still successful several advanced practical solutions are available to protect the systems, however these techniques have to keep up with the new challenges. One of the most-frequently applied ASLR bypass methods is guessing increasing the entropy of the Address Space Layout Randomization [26], it is a kind of mitigation, since it decreases the chance of a successful, brute-force, guessing attack. Forcing ASLR is another way to achieve better protection. Microsoft tried to prevent 0day exploitation with the Enhanced Mitigation Experienced Toolkit (EMET) [27] that provided some special advanced protections such as the anti-ROP technique. In 2016 Microsoft admitted that EMET is not proper for preventing 0day exploits and abandoned further development efforts. Microsoft has also introduced some new protections for the Edge browser [28] in 2016 such as the separated heap for the html objects or the delayed free to prevent the exploitation of use-after-free bugs. Other products, such as the Palo Alto exploit prevention [29], provides a wide choice of different protections, such as, detection of heap spraying and detection of ROP.

Since the main intension is stop the ROP-like exploitation several ideas are about to maintain and verify the correct control flow of a software [30]. One of the main questions of the protection over the efficiency is the performance. It is quite unfavorable if the exploit prevention comes with a performance penalty and slows down the execution speed significantly. Similarly, to DEP one good direction from performance point of view can be to provide hardware assisted anti-ROP protection. Such a solution is the Intel's Control Flow Enforcement (CFE) [31] which is a very promising technology.

CFE provides two components for the protection: the shadow stack and the indirect jump verifier. The shadow stack is a not accessible data storage place, where the copy of the method return pointers are placed during runtime. Each time a method exists, it obtains the return pointer from the normal data stack and the shadow, then the two return addresses are compared as a control. With this technique the execution of small code gadgets, with unintended *ret* instructions is prevented. The indirect jump verifier is a procedure which controls the indirect jumps during the code execution. The idea is to mark each legitimate indirect jump instruction with a *nop-like* special instruction. Whenever an indirect jump is executed this special *nop-like* instruction must follow it. If an unintended indirect jump is executed, the operating system can observe it.

Even this protection seems to be impossible to bypass and some new designs have already arisen, that have the potential to bypass it.

3 Analysis of Control Flow Enforcement Bypassing Exploitations

Control flow integrity protections such as the Intel's Control Flow Enforcement are promising plans to stop Return Oriented Programming without any speed decrease. The main question from software vulnerability exploitation point of view is still whether the software bug exploitation will be stopped or significantly decreased by making *ROP-like* techniques totally impossible or is it just a step of the exploitation-protection fight that makes exploitation techniques even more sophisticated. There are several ongoing research projects on new software vulnerability exploitation methods, such as, the Loop Oriented Programming [32] or the Data Oriented Programming (DOP) [33] and also the Counterfeit Object-oriented Programming (COOP) [34].

The main engine of the LOP is the loop gadget. The loop gadget is a special code fragment that realizes a loop and calls a method with indirect call instruction in each step. Figure 6 illustrates some theoretical examples of possible X86 loop gadgets:

```
1 mov esi, [edi]
2 add edi, 4
3 call esi
4 jmp 1
```

```
1 add esi, 4
2 call [esi]
3 jmp 1
```

Figure 6
Minimal loop gadgets

In the two presented cases the codes contain a loop and the instructions inside are repeated infinitely. Similarly, to JOP there is a register (*edi* in the first case and *esi* in the second example) which points to a memory (dispatcher table) and the pointer is moving to the next table entry in each step of the loop by the *add* instruction. The gadgets also contain an indirect *call* and that is how the functional gadgets are executed by reading the next address from the dispatcher table in every step. A better loop gadget is presented in Figure 7. This code fragment not only executes the functions in the dispatcher table but has a condition to quit from the loop and finish the program.

Since every functional gadget is a whole legitimate function, there is no shadow stack being compromised. Since each *ret* instruction has the *call* instruction pair, thus every *ret-like* instruction will be legitimate. From the functional gadgets point of view LOP has strict limitations. To bypass CFE only whole functions can be used as functional gadgets and especially only those methods which have the indirect jump marker at the beginning. Satisfying all these conditions CFE cannot prevent LOP execution, since the stack return pointer is not compromised and all the indirect jumps are legitimate. Figure 8 shows the control flow of LOP.

```
1 mov edi, edi          8  mov eax, [esi]
2 push ebp              9  test eax, eax
3 mov ebp, esp          10 jz 12
4 push esi              11 call eax
5 mov esi, [ebp+8]      12 add esi, 4
6 cmp esi, [ebp+0ch]    13 jmp 6
7 jnb 14                14 ...
```

Figure 7
Loop gadget in msvcr.dll [32]

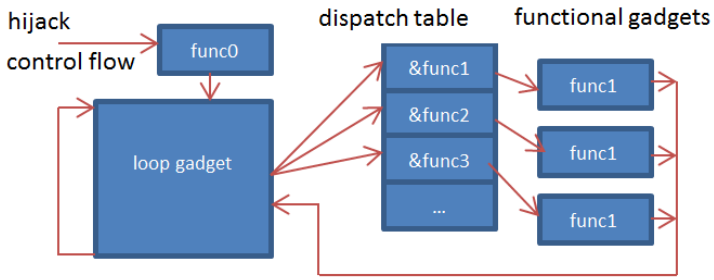

Figure 8
Loop Oriented Programming [32]

In the case of DOP [33] the main exploitation engine is the gadget dispatcher. Similarly, to the previous code-reuse techniques (JOP, LOP) a special code part controls the whole payload execution. The gadget dispatcher also has a loop but the functional gadgets are invoked in a different way than in case of LOP. DOP operates with six functional gadget types, but they implement different types of instructions: arithmetic/logical operation, assignment, load, store, jump, conditional jump. These functional gadget executions are repeated in various order during the payload execution with different parameters. The gadget dispatcher has a selector which sets which functional gadget should run in the next step and also sets the parameter of the next functional gadget. Figure 9 shows the control-flow of DOP.

Since the DOP functional gadgets implement general tasks, the gadget dispatcher of the DOP has more tasks than the LOOP gadget. It does not only invoke the next functional gadget, but sets the right parameters for the execution by customizing the input for the functional gadget. A practical example of a DOP gadget dispatcher is presented in Figure 10.

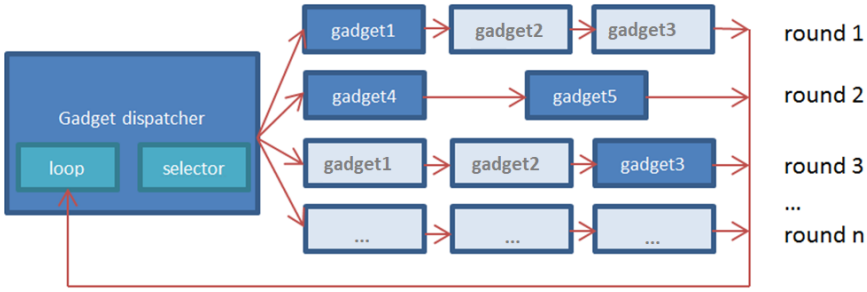

Figure 9
Data Oriented Programming [33]

```
1  struct server{ int *cur_max, total, typ;} *srv;
2  int connect_limit = MAXCONN; int *size, *type;
3  char buf[MAXLEN];
4  size = &buf[8]; type = &buf[12];
5  ...
6  while (connect_limit--)  {
7      readData (sockfd, buf);              // stack bof
8      if (*type == NONE )  break;
9      if (*type == STREAM)                 // condition
10         *size = * (srv->cur_max);        // dereference
11     else  {
12         srv->typ = *type;                // assignment
13         srv->total += *size;             // addition
14   } ... (following code skipped) ...
15 }
```

Figure 10
Example gadget dispatcher of Data Oriented Programming [33]

The Counterfeit Object Oriented Programming [34] is based on the virtual method calls of the Object Oriented Programming. Because of the inheritance, the object class is determined runtime in the case of virtual method call execution and the method addresses for each objects are stored in *vtable* structures. If the attacker manages to redirect the execution to a special virtual function, called the *main loop*, then they will be able to provide parameters to execute a Turing complete program without violating the Control Flow Enforcement. In the case of COOP, the dispatcher is the main loop and similarly to other loop techniques the task is to execute the functional gadgets in the right order and with the right parameters. The main loop as well as the functional gadgets are all legitimate virtual methods, so they are no longer really gadgets but long legitimate code parts. Figure 11 represents a main loop candidate, which is a destructor.

```
virtual ~Course( ) {
    for ( size_t  i = 0; i < nStudents; i++)
        students [i] ->decCourseCount ( );
    delete students;
}
```

Figure 11
A possible main loop for COOP [34]

As the attacker can set the *nStudent* parameter and the address pointing to the *student's* array, with the appropriate stack arrangement they can execute an arbitrary payload. Figure 12 shows a possible arrangement of the stack [34]. In Figure 12 the *students* array points back to the stack so the attacker can set the number of virtual methods to be executed, the address of the virtual methods to be executed, the order of the methods and also the method parameters.

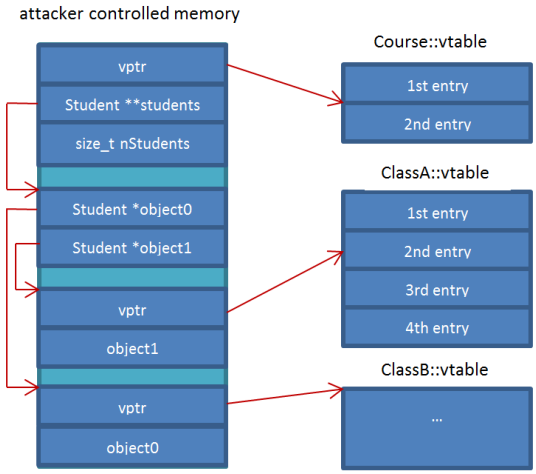

Figure 12
Stack arrangement for COOP [34]

Figure 13 shows the execution flow of COOP: taking advantage of a vulnerability the attacker redirects the code execution to the main loop and sets the stack pointer to a place where the main loop parameters are placed previously. COOP seems to be a very powerful technique against CFE as most of the programs currently use OOP.

According to our analysis it is important to distinguish between three different techniques considering the evolution of software vulnerability exploitation: In the first group we classify the techniques where the attacker can place and execute his own payload, like stack overflow, or classical use after free exploitation. Our second group contains the normal code reuse techniques, where the attacker executes the already existing code parts of the virtual memory, assembling the payload from small code parts that are not necessarily intended instructions called

the gadgets. We call that group *ROP-like* techniques. Our third group contains the latest exploitation techniques where the payload is assembled from legitimate functions and the execution is controlled by a code part containing a loop. We refer this technique as *LOP-like* techniques. Table 1 contains a summary of the different techniques and their main techniques of incursion.

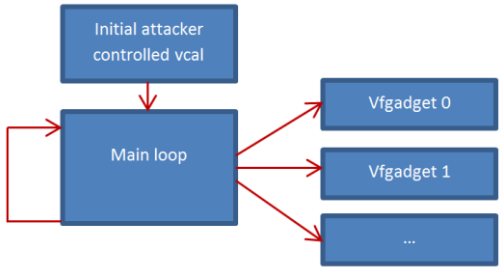

Figure 13
Counterfeit Object Oriented Programming [34]

Table 1
Software exploitation techniques

	Classical techniques	**ROP-like techniques**	**LOP-like techniques**
Method of payload execution	The payload to be executed is placed directly by the attacker	The payload is consist of small code parts (gadgets) from the virtual address space	The payload consist of legitimate methods from the virtual address space
DEP bypass	No	Yes	Yes
ASLR bypass	Not necessary	With additional vulnerability or memory leak	With additional vulnerability or memory leak
Shadow stack verification bypass	Stack overflow: No	ROP: No JOP: Yes	Yes
Indirect jump verification bypass	Use after free: No	ROP: Yes JOP: No	Yes
CFE bypass	No	No	Yes
Turing completeness	Yes	Yes, but depends on the gadget catalog	Yes, but depends on the method catalog
Example techniques	Stack overflow, use after free	ROP, JOP	LOP, DOP, COOP

The latest exploitation techniques are definitely able to bypass the Control Flow Enforcement technique [31]. So it is clear that if CFE will be used in the future then attackers will turn to *LOP-like* techniques. On the other hand, it is important to mention that there is no practical experience on the usability of these techniques.

Table 2
Control flow bypassing exploitations

	Loop Oriented Programming	**Data Oriented Programming**	**Counterfeit Object Oriented Programming**
Control gadget name	Loop gadget	Gadget dispatcher	Main loop
Control gadget functionality	Calls the methods step by step according to the dispatcher table	Selects the type of function first and call them step by step	Calls the virtual methods step by step with their parameter according to the stack arrangement
DEP bypass	Yes	Yes	Yes
ASLR bypass	With additional vulnerability or memory leak	With additional vulnerability or memory leak	With additional vulnerability or memory leak
Shadow stack verification bypass	Yes	Yes	Yes
Indirect jump verifier bypass	Yes	Yes	Yes
CFE bypass	Yes	Yes	Yes
Turing completeness	Yes, but depends on the method catalog	Yes, but depends on the method catalog	Yes, but depends on the virtual method catalog

LOP-like techniques have to satisfy three conditions according to our analysis:

1) The virtual address space must have proper *loop-like* gadget
2) Possibility to redirect the code execution to the loop with the appropriate parameters
3) Appropriate method catalogs to execute the desired payload

The first and the third conditions are influenced by the content of the virtual address space. The second condition is influenced by the type of the vulnerability, as well as, the characteristics of the *loop-like* gadget dispatcher. Table 2 summarizes and compares the main behavior of the LOP-like exploitation methods. As it can be seen in Table 2 all three methods use a very similar idea: There is a loop which gets the control by a vulnerability with an initial setting. Then the loop continuously invokes legitimate methods from the virtual address space according to the previously placed method table and parameters by the

attacker. However there is no hardware assisted Control Flow Enforcement yet, but the presented three exploitation techniques seem to be a real option to bypass CFE.

According to our analysis, preventing such an attack type is only currently possible during compilation time. From the point of view of the requirements the following things would be necessary to avoid such exploitations:

1) The key element of the exploitation is the loop-like gadget. The compilers should check and at least provide a warning message if a loop like gadget is available after the compilation.
2) Avoiding unwanted code redirection would be a basic prevention, but considering the current state this cannot be guaranteed. Almost all type of software vulnerabilities can achieve unwanted control flow change. Since software vulnerabilities cannot be totally excluded the prevention cannot be built on this either.
3) Preventing the creation of dispatcher table is also not realistic. With OOP different user controlled objects can be created in the heap. The only thing that is necessary from the attacker's point of view is to place the dispatcher table in a predictable place. This can be carried out together with a memory leak.

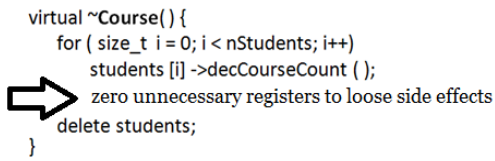

Figure 15
Loosing side effects of virtual methods

According to our analysis the only option to prevent such exploitations, is to prevent the loop like gadget compilation. On the other hand, in some cases, such as, in Figure 12, the loop like code block was created on purpose (iterating through the students). In such cases, our suggestion is to append the code and zero all registers, except for the return value in each step of the loop (Figure 15). With this solution the virtual methods negate the unwanted side effects that the attacker can use in these exploitations.

Conclusions

Based on previous experiences, we cannot simply let system security be based on the assumption of having perfect software, without vulnerabilities, to avoid software vulnerability exploitations. Additional advanced protections are necessary. From a performance point of view, hardware based solutions are preferred, such as DEP. However, ROP, which is the most popular technique of today's exploitations, can bypass DEP. Control Flow Integrity techniques, such as,

CFE, aim to prevent *ROP-like* techniques, but new exploitation ideas, such as, LOP, DOP or COOP, have appeared recently. In this study, the main stages of software bug exploitations are analyzed, with a special focus on the behavior and capabilities of the cutting-edge techniques. We conclude, currently, it is not clear if there is any protection that is capable of stopping the exploitation of unknown software bugs; the best thing that can be done on the protection side, is to mitigate the potential for successful exploitation.

To avoid *LOP-like* exploitations, we suggested possible solutions to mitigate the risk of such attacks. According to our analysis the most feasible way of preventing loop oriented programming type attacks, can be implemented during the compilation stage. With code blocks presented in Figures 7 and 8, the compiler should try to avoid them, or at least provide a warning message if such code is created. For other loop like code blocks, such as, in COOP the compiler should try to insert extra code, that force the virtual methods to negate the side effects. With these added instructions, the attackers would not be able to create useful gadget chains.

References

[1] Offensive Security. Offensive securitys exploit database archive. https://www.exploitdb.com/

[2] Exploitalert website. http://exploitalert.com

[3] Blogger technology. Metasploit. https://blgtechn.blogspot.no/2012/08/metasploit.html

[4] Microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003, https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-dataexecution-prevention-dep-feature-in-windows-xp-service-pack-2-windows-xp-tablet-pcedition-2005-and-windows-server-2003, 2006

[5] R. Seka Lixin Li, James E. Jus. Address-space randomization for windows systems. http://seclab.cs.sunysb.edu/seclab/pubs/acsac06.pdf, 2012

[6] B. Pak. Microsoft edge (Windows 10) - 'chakra.dll' info leak / type confusion remote code execution. https://www.exploit-db.com/exploits/40990/, 2017

[7] D. Dörr. Drupal 7.32 - sql injection (php), 2014, https://www.exploit-db.com/exploits/34993

[8] Cve details - the ultimate security vulnerability datasourse. http://cvedetails.com

[11] E. Levy. Smashing the stack for fun and profit. Phrack Mag, 49(14), 8 1996

[12] M. Kaempf. Smashing the heap for fun and profit. Phrack Magazine, 57(11), 8 2001

[13] Scut / team teso. Exploiting format string vulnerabilities. https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf, 2001

[14] CWE Common Weakness Enumeration. Cwe-416: Use after free. https://cwe.mitre.org/data/definitions/416.html, 2012

[15] P. M. Wagle. Stackguard: Simple buffer overflow protection for gcc. In Proceedings of the GCC Developers Summit, pp. 243-256, 2003

[16] J. N. Ferguson. Understanding the heap by breaking it. http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf

[17] Microsoft. Preventing the exploitation of structured exception handler (seh) overwrites with sehop. https://blogs.technet.microsoft.com/srd/2009/02/02/preventing-the-exploitationof-structured-exception-handler-seh-overwrites-with-sehop/, 2009

[18] S. El Sherei. Return to libc. https://www.exploit-db.com/docs/28553.pdf

[19] H. Shacham, E. Buchanan, R. Roemer, and S. Savage. Return-oriented programming:Exploitation without code injection. https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

[20] T. Bletsch, X. Jiang, and V. Freeh. Jump-oriented programming: A new class ofcode-reuse attack. In 17th ACM Computer and Communications Security, 2010

[21] E. Bosman and H. Bos. Framing signalsa return to portable shellcode. In SP '14 Proceedings of the IEEE Symposium on Security and Privacy, pp. 243-258, 2014

[22] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. https://people.eecs.berkeley.edu/daw/papers/rop-usenix14.pdf, 2014

[23] H. Shacham, M. Page, B. Pfaff, Eu-Jin Goh, N. Modadugu,and D. Boneh. On the effectiveness of address-space randomization. http://benpfaff.org/papers/asrandom.pdf, 2004

[24] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. http://www.scs.stanford.edu/sorbo/brop/bittau-brop.pdf, 2015

[25] L. Davi, C. Liebchen, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In NDSS Symposium 2015, 2015

[26] Ars Technica. Firefox 0-day in the wild is being used to attack tor users. https://arstechnica.com/information-technology/2016/11/firefox-0day-used-against-tor-users-almost-identical-to-one-fbi-used-in-2013/, 2016

[27] A. Kondratenko. Cve-2017-3881 cisco catalyst rce proof-of-concept. https://artkond.com/2017/04/10/cisco-catalyst-remote-code-execution/. 2017

[28] K. Johnson and M. Miller. Exploit mitigation improvements in windows 8. https://media.blackhat.com/bh-us-12/Briefings/M_Miller/ BH_US_12_ Miller_Exploit_Mitigation_Slides.pdf

[29] Microsoft. The enhanced mitigation experience toolkit. https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit, 2012

[30] M. V. Yason. Understanding the attack surface and attack resilience of project spartans (edge) new edgehtml rendering engine. https://www.blackhat.com/docs/us-15/materials/us-15-Yason-Understanding-The-Attack-Surface-And-Attack-Resilience-Of-Project-Spartans-New-EdgeHTML-Rendering-Engine-wp.pdf, 2015

[31] Paloalto Networks. Traps administrators guide. https://www.paloaltonetworks.com/documentation/33/endpoint/endpoint-admin-guide, 2017

[32] J. Tang. Exploring control flow guard in windows 10. http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf, 2016

[33] Intel. Control-flow enforcement technology preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcementtechnology-preview.pdf, 2016

[34] Y. Li, B. Lan, H. Sun, C. Su, Y. Liu, and Q. Zeng. Loop-oriented programming: A new code reuse attack to bypass modern defenses. In 2015 IEEE Trustcom/BigDataSE/ISPA, pp. 91-97, IEEE Computer Society

[35] H. Hu, S. Shinde, S. Adrian, Z. Leong Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. https://www.comp.nus.edu.sg/ ~shweta24/publications/dop_oakland16.pdf

[36] F. Schuster, T. Tendyck, C. Liebcheny, L. Davi, A. Sadeghiy, and T. Holz. Counterfeit object-oriented programming- on the difficulty of preventing code reuse attacks in c++ applications. syssec.rub.de/media/emma/veroeffentlichungen/2015/03/28/ COOP-Oakland15.pdf, 2015