Exploit Prevention, Quo Vadis?

László Erdődi(✉) and Audun Jøsang

University of Oslo, Oslo, Norway
{laszloe,josang}@ifi.uio.no

Abstract. Exploits are advanced threats that take advantage of vulnerabilities in IT infrastructures. The technological background of the exploits has been changed during the years. Several significant protections have been introduced (e.g. Data Execution Prevention, Enhanced Mitigation Experience Toolkit, etc.), but attackers have always found effective ways to bypass any protection. This study gives a summary on the main software vulnerability exploitation methods including protections. Furthermore the study analyzes the capabilities and the predicted future of software exploitation in the light of the new protection technologies.

Keywords: Exploits · Prevention · Vulnerability · Control-flow · Protection

1 Introduction

According to a common definition, an exploit is a piece of software, a chunk of data, or a sequence of commands that takes advantage of a bug or vulnerability in order to cause unintended or unanticipated behavior to occur in computer software [35]. From the vulnerability point of view two major categories can be distinguished: the configuration error based and the software error based exploits. The object of this paper is to analyze the case when the software code contains vulnerability both from the attack and the protection point of view. Within this the emphasis is laid on the lower level type of vulnerabilities where the exploitation is carried out directly in the virtual memory.

Exploits are usually categorized from different aspects such as the capability (e.g. remote code execution, DOS), the way of execution (local, remote) or the platform it can be applied for (Windows, Linux, Ios, etc.). The exploit database [25] is a website where users can submit ready to use exploits. Even if this site obviously does not contain all the existing exploits it is nevertheless interesting to observe the evolution in the number of the available exploits throughout the years. The number of new exploits was on the top around December 2009. After this a significant decrease can be observed. The reason for the decline can be both the use of new protections such as the Data Execution Prevention (DEP) [18] or the Address Space Layout Randomization (ASLR) [17] and other new possibilities such as the dark web that appeared for exploit writers. An additional

© Springer International Publishing AG 2017
G. Livraga and C. Mitchell (Eds.): STM 2017, LNCS 10547, pp. 180–190, 2017.
DOI: 10.1007/978-3-319-68063-7_12

exploit database is the Metasploit framework [32] which makes exploits available in a unified form providing an easy–to–use framework for the exploits.

The Common Vulnerabilities and Exposures database [6] is an alternative source of information about available exploits. It is important to consider the dark web communities which offer exploits for virtual currencies. An exploit is usually connected to one particular vulnerability on a particular software, but there are some exceptions. Several exploitation and attacking techniques exist and considering the protection the main focus is to stop the exploitation without significant resource usage overhead. Hardware based techniques as protection are usually more preferable since they hardly slow down the normal execution speed. Section 2 focuses on the different exploitation and protection techniques, while in Sect. 3 the current situation and future predictions are analyzed. In Sect. 4 the latest potential exploitation techniques are analyzed.

2 Exploitation and Protection Techniques

2.1 Early Vulnerability Exploitations

In the early years of exploitations the attacker did not have to focus on bypassing any protection that was provided either by the compiler or the operating system. In the virtual memory everything was allocated and applied for the sake of the fast and efficient code execution. The program code is loaded into the virtual memory as well as the shared libraries with the operating system API. Each thread has its own stack segment where the methods data are placed by stack frames. The whole process has some common heaps, where the dynamically allocated objects are stored. The objects have virtual method tables where the actual addresses of the virtual methods are placed. The heap is organized as series of linked list chunks because of the effective and fast memory allocation and free in runtime. Figure 1, shows the arrangement of the virtual memory.

In the early stages the security of the software is based only on the code security. Unfortunately with a single coding error the attacker can simply misuse the software to execute malicious code. This is possible with several well-known techniques such as the stack overflow [16], the heap overflow [14], the format string vulnerability [24] or the use–after–free bug [8], etc. In the case of stack overflow [16] a local variable (e.g. a string or an array) is overwritten in the stack frame. As the stack frame contains the method return pointer the attacker can redirect the execution to an arbitrary place by providing a new return pointer. By placing the attack payload in the corrupted local variable on the stack the attacker redirects the execution to the stack itself and the payload is executed there. In case of heap overflow [14] the overwritten variable is in the heap. By overrunning the current heap chunk the attacker is able to modify the next heap chunk header data such as the addresses pointing to the next and previous chunk. When the attacker-modified chunk is freed the modified header pointers are used for merging the current chunk with other chunks. In that process the header pointers are used for writing data, so the attacker can write an arbitrary data to an arbitrary place. That is how the control flow is modified to execute

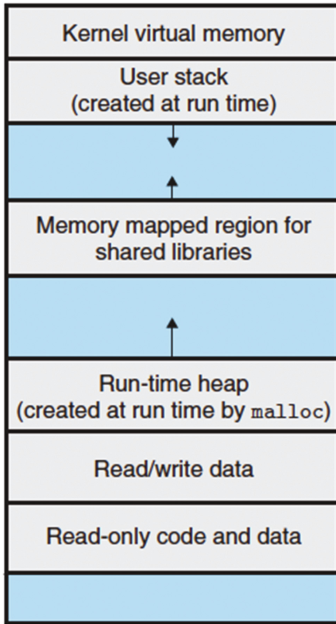

Fig. 1. Virtual address space layout

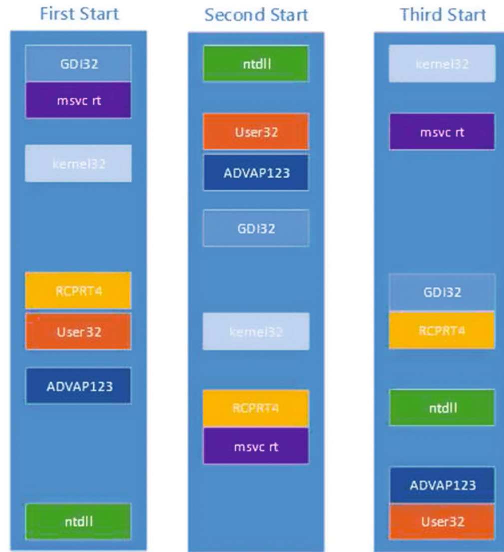

Fig. 2. Address Space Layout Randomization [17]

the malicious code. In case of format string vulnerability [24] the attacker can write an almost arbitrary data to an arbitrary place by providing special invalid string formatting parameters. By overwriting a stack method return pointer or modifying 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 [8] is based on the virtual method table modification. If an object is used after being freed then the attacker can allocate a fake object to the same place where the original object was. The objects contain a pointer to its own virtual address table. In the case of the fake object that value is pointing to an attacker-created virtual address table with pointers to the malicious code (Fig. 2).

2.2 Early Defenses

The early protections focused on protecting the critical data from the right program-flow point of view. Stack cookie [34] is an example technique to protect the method return address from being overwritten. Since the stack cookie is placed between the method local variables and the return pointer, any modification outside the local variables results in the change of the stack cookie. This modification signs the stack frame corruption for the operating system. Although this protection is quite good to filter the stack frame corruption, it

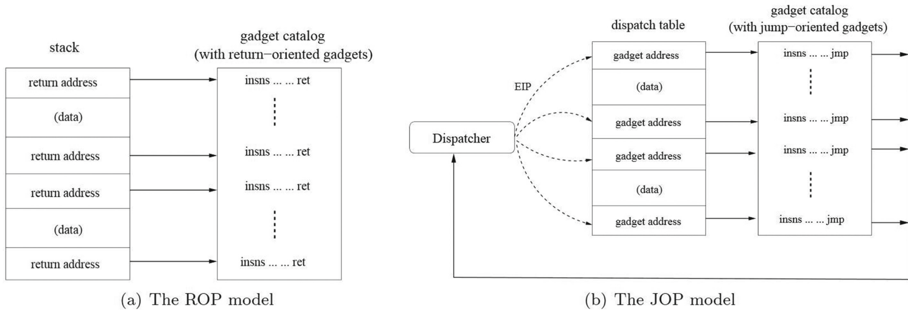

Fig. 3. Return Oriented and Jump Oriented Programming [17]

comes with a significant performance penalty. The heap chunk header modification is prevented by the secure unlink process [10] that validates the chunk header pointers before it is merged with another chunk. Another protection is the secure structured exception handling [19] which validates the exception handler pointer before it is executed. Several robust protections appeared in the middle of the 2000s. These protections such as the Data Execution Prevention (DEP) [18] and the Address Space Layout Randomization (ASLR) [17] do not aim to prevent only one typical exploitation, but the aim was to make the exploitation more difficult in general. Data execution prevention enforces memory page rights for the different types of pages. Reading the page data, writing the page data or executing the page data are all different types of operations. DEP ensures that a memory page cannot be written and executed at the same time. Using this protection several previously mentioned exploitation methods are disabled since the payload can be written to a writable memory place but it cannot be executed. Address Space Layout Randomization [17] is about to prevent malicious code reuse. If the place of the virtual memory pages are randomized every time when the program is launched (Fig. 3) then the attacker cannot rely on the known memory addresses. In the early protection stage of the ASLR the randomization entropy was insufficient to protect the software against address guessing.

2.3 Advanced Exploitations

After Data Execution Prevention [18] had been widely implemented, exploit writers had to turn to new techniques. Since the attacker could no longer place the payload to execute it because of the DEP, the main idea became to execute the already existing code parts that have the right to be executed. The first technique was the return to libc [28] type of exploitations where the corrupted method is redirected to an operating system API method such as the `WinExec` or `Execve` methods. In this case the attacker only provides the method parameters (e.g. the name of the software that has to be executed). However this technique is only capable to execute only one method, but choosing the right method with right parameters it can be sufficient. A huge break-through was the invention of

the Return Oriented Programming [26]. This technique assembles the payload from small code parts called the gadgets. As the gadgets are the part of the code libraries in the virtual memory, this is a very sophisticated code reuse technique. A gadget contains some assembly instructions with a `ret` type of instruction at the end. Considering the previously mentioned stack overflow case, the attacker has to place the series of gadget addresses on the corrupted stack frame. When the corrupted method exits, the execution is directed to the first gadget. Because of the `ret` instruction at the end of the gadget the execution is directed to the next gadget by taking the next address on the corrupted stackframe by the `ret` instruction. Since ROP is Turing complete the limitation of ROP highly depends on the gadget catalog provided by the virtual address space. Practically there is no limitation, the attacker can always have enough gadgets to turn off the DEP and continue the payload execution in the traditional way. A generalization of ROP is the Jump Oriented Programming (JOP) [3]. Similarly to ROP, JOP executes the payload step by step by using small code parts called 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 dispatcher gadget maintains a table pointer to execute the functional gadgets after each other in the right order. Instead of building upon the stack and the `ret` instruction, JOP realizes its own stack like structure the dispatcher table and the concatenation of the gadgets are ensured by the dispatcher gadget and not the `ret` like instructions.

Several other forms of scattered code reuse technique exist such as the Sigreturn Oriented Programming (SROP) [4] or the Call Proceeded Return Oriented Programming (CPROP) [5]. In the first case the exploitation is based on the kernel context switching which saves the current execution context in a frame on the stack. Unlike ROP, SROP exploits are usually portable across different binaries and can bypass ASLR in some cases. Call Proceeded Return Oriented Programming uses whole functions as a gadget in order to bypass the control flow protections. Bypassing ASLR in code reuse attacks is always a challenge. Special techniques such as the Blind Return Oriented Programming (BROP) [2] and Just in Time Return Oriented Programming [7] can bypass ASLR by guessing the randomization offset or with just in time payload customization. In some cases ASLR can be bypassed by simple guessing the randomization offset [27] or by taking advantage on another vulnerability that expose the randomization offset [22].

2.4 Enhanced Protections

Due to the continuously improving exploitation techniques, protection methods have to keep up with the new challenges. Increasing the entropy of the Address Space Layout Randomization [13] decreases the chance to successfully bruteforce the randomization offsets. Forcing ASLR is another technique to achieve better protection. Microsoft aimed to prevent the exploitation with the Enhanced Mitigation Experienced Toolkit (EMET) [20] that provides special protections such as the anti-ROP technique. In 2016 Microsoft admitted that EMET is

not proper for preventing 0 day exploits and stopped the development of it. Microsoft has also introduced some new protections for the Edge browser [36] such as the separated heap for the html objects or the delayed free to prevent the exploitation of use-after-free bugs. Other software such as the Palo Alto exploit prevention [21] provides wide choice of different protections e.g. detection of heap spraying, detection of ROP, etc. Several other ideas exist to maintain and verify the correct control flow of a software [29]. One of the main questions of the protection is the performance. It is unfavorable if the exploit detection slows down the execution speed significantly. Hardware based protection ideas such as the Intel's Control Flow Enforcement (CFE) [12] are very promising technologies. According to CFE the protection is provided by two components: the shadow stack and the indirect jump verifier. CFE maintains two separate stacks: the data stack for the normal operation, but also a shadow stack which is not accessible for the code. Whenever a method returns both the data stack and the return stack pointers are popped and compared as a control. This technique should prevent the execution of small gadgets with not intended `ret` instructions. The indirect jump verifier is a method which controls the indirect jumps with a `nop-like` special instruction. Whenever an indirect jump is executed this special `nop-like` instruction must follow it. This measure should stop the unintended indirect jumps through the code libraries.

3 Current Exploits

Although several protections exist, exploits still represent a real danger for IT systems. Nowadays attackers have to consider the DEP and the ASLR as a basic feature of the modern operating systems, so bypassing them is essential from the successful exploitation point of view. Some browser exploits turned into light in the late 2016 and the favorite exploitation technology was the Just in time Return Oriented Programming. At the end of 2016 a Firefox/Tor exploit (CVE-2016-9079) is revealed [31] which attacked Tor browser users. The exploit maps the Windows PE 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 [22]. 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. This case belongs to the Just in Time Return Oriented Programming category as well as well as the case of the Tor exploit. ROP based exploits are deployed against network devices too. A vulnerability (CVE 2017-3881) [15] in the Cisco Cluster Management Protocol (CMP) processing code in Cisco Software could allow an unauthenticated, remote attacker to execute code with elevated privileges. The exploit for this vulnerability uses ROP to bypass the DEP protection as well.

Considering other cases too, it is clear that the main technique of the current exploits is still the Return Oriented Programming. DEP and ASLR thought to

be a very strong protection together, but current examples show that they can be bypassed routinely in several cases. That is the reason why the current direction of the protection strengthening is to enforce the right control-flow in order to disable ROP. For example, Intel's Control Flow Enforcement is a promising plan that should stop Return Oriented Programming without any speed decrease. The question is, if the software bug exploitation will be stopped or significantly decreased by making ROP totally impossible with some countermeasures or is it just a step of the exploitation-protection fight that makes exploitation techniques more sophisticated. Currently it is very difficult to predict what is going to happen after the protection against ROP is completely solved. There are several ongoing research projects on new exploitation methods such as for example the Loop Oriented Programming [1] or the Data Oriented Programming (DOP) [11] and also the Counterfeit Object-oriented Programming (COOP) [23]. The following chapter focuses on these types of exploitations these can be one of the next milestones of the modern software exploitation.

4 Bypassing the Control Flow Enforcement - The New Direction

Current protections and tendencies indicate that code reuse will be the main technology in the future too. Bypassing ASLR is possible with brute-forcing and through information leakage today, but more sophisticated ASLR bypass techniques [9,33] are already presented. Considering a successful ASLR bypass, Loop Oriented Programming [1] looks like a possible option against control flow enforcement. The most important part of the LOP is the loop gadget. The loop gadget executes legitimate shared library methods after each other to carry out the malicious task. Similarly to JOP, the loop gadget is like a dispatcher. It concatenates the functional gadgets, so the payload is executed step-by-step. Contrary to ROP and JOP the LOP gadgets are not only small code parts. Each functional gadget is a legitimate method, so the shadow stack protection is useless against it, because exiting from a method does not violate the regular method exit rules. Since Control Flow Integrity also has the protection against indirect jumps, each functional gadget must contain the indirect jump marker `nop` like instruction at the very beginning of the gadget. This can decrease the gadget catalogue significantly, which is a challenge for exploit writers. Figure 4, shows the LOP execution process [1].

According to our analysis the following conditions have to be satisfied in order to carry out a successful LOP exploitation: 1. Bypassing ASLR in order to use accurate memory addresses; 2. Having an attacker controlled memory region that contains the gadget method addresses that have to be executed in the right order; 3. Having the appropriate loop gadget that implements the loop for the execution while reading the address table and directing the execution to the appropriate method; 4. Initializing the attack successfully by directing the execution to the loop gadget with the appropriate register values; 5. Having sufficient method catalog to turn off the DEP and continue the payload execution in the conventional way.

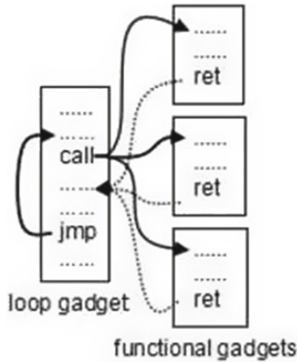

Fig. 4. Loop Oriented Programming

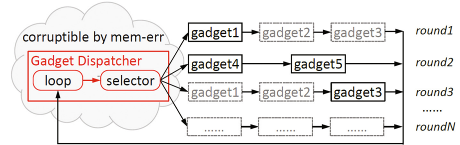

Fig. 5. Data Oriented Programming [11]

Since there are already existing options to bypass ASLR and there is no predictable solution to prevent ASLR bypass by information leakage, we consider that the first condition can be satisfied. Having of an attacker controlled method address table can be satisfied easily too since attackers can easily write data to the virtual memory e.g. with heap spraying [30]. According to some previous analyses [1], finding appropriate loop gadgets is also possible. Having a sufficient method catalog (condition 4) is a question as well as the successful initialization of the attack (condition 5). Since processors with hardware supported Control Flow Enforcement do not exist yet, there is nothing that can be stated related to the LOP gadget catalog. With Intel's proposed Control Flow Enforcement the libraries should be rewritten with the indirect jump protection instructions. So condition 5 will depend on the new CFE supported libraries. Condition 4 is significantly influenced by the type of the vulnerability. As the exploitation has to direct the instruction pointer to the loop gadget with the appropriate register settings, if the loop gadget exists and the attacker can set the method table index of the loop gadget, so can the initialization be successful. It is clear that conventional stack overflow cannot be used with CFE anymore with the return address modification, but if the vulnerable method contains an indirect call where the address is read from the corrupted stack then the exploitation can be successful. In the case of heap related vulnerabilities e.g. in use–after–free, the initialization can be successful if the attacker can control at least two registers: the one that contains the loop gadget method address and another one with the method table addresses.

Data Oriented Programming [11] seems to be another option to bypass Control Flow Enforcement. Similarly to JOP and LOP the code execution is controlled by one special code part, in this case this is the gadget dispatcher (Fig. 5). The gadget dispatcher has a loop which is controlled by the attacker. The loop contains different type of data oriented gadget invocations such as assignment, store, load and jump. As the attacker controls the local variables of the corrupted function he can set how many times the loop is executed and also the

loop parameters in every step (which data oriented gadget should be invoked with which parameters). With the loop the appropriate data oriented gadgets are chained.

In case of COOP [23], virtual functions exist in the vulnerable application are repeatedly invoked on special C++ objects carefully arranged by the attacker. These special C++ objects are injected by the attacker and contain an attacker-chosen virtual pointer and a few attacker-chosen data fields. Similarly to other code reuse attacks chaining different code parts are directed by a special gadget: COOP program essentially relies on a special main loop containing a virtual function call.

From the protection point of view the most promising technology is the Control Flow Enforcement. Hardware supported CFE does not exist yet, but it is clearly visible that operating systems has to apply new libraries to support hardware assisted control flow enforcement. Our analysis concluded that the control flow enforcement solutions will make the successful exploitation more difficult: 1. In all of the existing control flow bypassing techniques the attacker has to control more parameters (registers, local variables) than in the case of an average exploitation today; In all of the existing control flow bypassing techniques the attacker has to identify a special code part (loop gadget, gadget dispatcher, etc.) which is responsible for chaining the payload parts from small code blocks. According to our investigations the possible control flow bypassing techniques must be considered when overwriting the operating system libraries. This should be done by identifying dangerous code parts (loop gadget candidates, gadget dispatcher candidates) and remove them with the introduction of hardware assisted control flow enforcement libraries.

5 Summary

Based on previous experiences we cannot simply let system security be based on perfect software without vulnerabilities to avoid software vulnerability exploitations. Additional advanced protections are necessary. From performance point of view hardware based solutions are preferred such as the DEP. However the ROP which is the favorite technique of todays exploitations can bypass DEP. ASLR is an efficient protection against ROP, but information leakage can reveal the randomization offset, which makes the code reuse type of exploitations still possible. Anyhow, the trend in protection innovation indicates that sooner or later ROP based exploits will be disabled with some protection technique. CFE aims to prevent ROP, for example. New exploit ideas such as the LOP, DOP and COOP are being published continuously to bypass the new protections. Right now it is not clear whether there exists any protection that is capable of stopping the exploitation of unknown software bugs or that the only thing that can be done on the protection side is to mitigate the percentage of successful exploitations. In this paper we summarized the main exploitation and protection techniques and in addition we analyzed the latest code reuse exploitations. These might be the most relevant exploitations of the future that underlines the question: Exploit prevention - Quo Vadis?

References

1. Li, Y., Lan, B., Sun, H., Su, C., Liu, Y., Zeng, Q.: Loop-oriented programming: a new code reuse attack to bypass modern defenses. In: 2015 IEEE Trustcom/BigDataSE/ISPA, pp. 91–97. IEEE Computer Society (2015)
2. Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., Boneh, D.: Hacking blind (2015). http://www.scs.stanford.edu/sorbo/brop/bittau-brop.pdf
3. Bletsch, T., Jiang, X., Freeh, V.: Jump-oriented programming: a new class of code-reuse attack. In: 17th ACM Computer and Communications Security (2010)
4. Bosman, E., Bos, H.: Framing signalsa return to portable shellcode. In: SP 2014 Proceedings of the IEEE Symposium on Security and Privacy, pp. 243–258 (2014)
5. Carlini, N., Wagner, D.: ROP is still dangerous: breaking modern defenses (2014). https://people.eecs.berkeley.edu/daw/papers/rop-usenix14.pdf
6. cvedetails.com. CVE details - the ultimate security vulnerability datasourse. http://cvedetails.com
7. Davi, L., Liebchen, C., Snow, K.Z., Monrose, F.: Isomeron: code randomization resilient to (just-in-time) return-oriented programming. In: NDSS Symposium 2015 (2015)
8. CWE Common Weakness Enumeration. CWE-416: use after free (2012). https://cwe.mitre.org/data/definitions/416.html
9. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over ASLR: attacking branch predictors to bypass ASLR (2016). http://www.cs.ucr.edu/nael/pubs/micro16.pdf
10. Ferguson, J.N.: Understanding the heap by breaking it (2007). http://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf
11. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-oriented programming: on the expressiveness of non-control data attacks (2016). http://ieeexplore.ieee.org/iel7/7528194/7546461/07546545.pdf
12. Intel. Control-flow enforcement technology preview (2016). https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf
13. Johnson, K., Miller, M.: Exploit mitigation improvements in Windows 8 (2012). http://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf
14. Kaempf, M.: Smashing the heap for fun and profit. Phrack Mag. **57**(11), 8 (2001)
15. Kondratenko, A.: CVE-2017-3881 Cisco Catalyst RCE Proof-of-Concept (2017). https://artkond.com/2017/04/10/cisco-catalyst-remote-code-execution/
16. Levy, E.: Smashing the stack for fun and profit. Phrack Mag. **49**(14), 8 (1996)
17. Seka, R., Li, L., Just, J.E.: Address-space randomization for windows systems (2012). http://seclab.cs.sunysb.edu/seclab/pubs/acsac06.pdf
18. 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 (2006). https://support.microsoft.com/en-us/help/875352/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
19. Microsoft: Preventing the exploitation of structured exception handler (SEH) overwrites with sehop (2009). https://blogs.technet.microsoft.com/srd/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/

20. Microsoft: The enhanced mitigation experience toolkit (2012). https://support.microsoft.com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit
21. Paloalto Networks. Traps administrators guide (2017). https://www.paloaltonetworks.com/content/dam/pan/en_US/assets/pdf/framemaker/32/endpoint/endpoint-admin-guide/section_1.pdf
22. Pak, B.: Microsoft edge (Windows 10) - 'chakra.dll' info leak/type confusion remote code execution (2017). https://www.exploit-db.com/exploits/40990/
23. Schuster, F., Tendyck, T., Liebcheny, C., Daviy, L., Sadeghiy, A.-R., Holz, T.: Counterfeit object-oriented programming - on the difficulty of preventing code reuse attacks in C++ applications (2015). http://syssec.rub.de/media/emma/veroeffentlichungen/2015/03/28/COOP-Oakland15.pdf
24. scut/team teso. Exploiting format string vulnerabilities (2001). https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf
25. Offensive Security. Offensive securitys exploit database archive. https://www.exploit-db.com/
26. Shacham, H., Buchanan, E., Roemer, R., Savage, S.: Return-oriented programming: exploitation without code injection (2008). https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf
27. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., Boneh, D.: On the effectiveness of address-space randomization (2004). http://benpfaff.org/papers/asrandom.pdf
28. El Sherei, S.: Return to libc. https://www.exploit-db.com/docs/28553.pdf
29. Tang, J.: Exploring control flow guard in Windows 10 (2016). http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf
30. Corelan Team: Exploit writing tutorial part 11: heap spraying demystified (2011). https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/
31. Ars Technica: Firefox 0-day in the wild is being used to attack tor users (2016). https://arstechnica.com/security/2016/11/firefox-0day-used-against-tor-users-almost-identical-to-one-fbi-used-in-2013/
32. Blogger technology: Metasploit. https://blgtechn.blogspot.no/2012/08/metasploit.html
33. van Schaik, S., Razavi, K., Gras, B., Bos, H., Giuffrida, C.: Reverse engineering hardware page table caches using side-channel attacks on the MMU (2017). http://www.cs.vu.nl/herbertb/download/papers/revanc_ir-cs-77.pdf
34. Wagle, P.M.: Stackguard: simple buffer overflow protection for GCC. In: Proceedings of the GCC Developers Summit, pp. 243–256 (2003)
35. Wikipedia. Exploit (computer security) (2010). https://en.wikipedia.org/wiki/Exploit_(computer_security)
36. Yason, M.V.: Understanding the attack surface and attack resilience of project spartans (edge) new edgehtml rendering engine (2015). 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