ATLANTIS PRESS

Advances in Computer Science Research, volume 59

7th International Conference on Education, Management, Computer and Medicine (EMCM 2016)

Eternal War in Software Security: A Survey of Control Flow Protection

Bowen Tang[1, a*], Huan Ying[2, b], Wei Wang[1, c] and Huabin Tang[1, d]

[1]Capital Normal University, Beijing 100048, China

[2] China Electric Power Research Institute, Beijing 100192, China

[a]tangbowen@ict.ac.cn, [b]yinghuan@epri.sgcc.com.cn,

[c]wangwei02@ict.ac.cn, [d]tanghuabin@ict.ac.cn

Keywords: Software security; Control flow protection; Operation system; Program analysis; Performance

Abstract. Software security is the cornerstone of computer system security. Among all the elements consisting of software security, control flow protection is undoubtedly the most important one. Once the process's control flow is hijacked, attacker can manipulate it to implement a variety of malicious acts and break through other protection mechanisms which ultimately lead to the control of the entire system. This paper will present a series of offensive and defensive technologies about Control Flow Protection which have been developed in the past three decades. The paper will elaborate the causes of their emergence, explain the principle of their implement, and compare the security and performance of their method. Additionally, it will introduce some other technologies applied in the progress of attack and mitigation, such as program analysis, virtual memory management, machine learning and so on. Through those above illustration and analysis, the paper summarizes three primary suggestions which not only can enlighten security engineers on the design of new methods, but also can help general developers to estimate their software's robustness, practicability and performance.

Introduction

Computer System is often subject to external attacks that aim to steal its private data or control its behavior. Typically, such attacks can be launched by web communication or by local social engineering. But neither remote nor indigenous attacks, they must be implemented by exploitation of low level software vulnerabilities. From the view of Trust Computing Base Theory, software security must occupy the central position of the whole computer security field [1]. However software security is just a unified concept which consists of many significant sub-instances, such as Data Privacy Protection [2], Digital Rights Management [3], Application Access Control [4] and so on. Among all of the elements, Program Control Flow Protection must be the most important.

Control Flow Hijack has been an oldest threat in software security for three decades because of its awesome capacity. If a running program's control flow is hijacked by attackers, it can manipulate all data in memory space and disguise as a benign client to compromise other defenses which may ultimately lead to controlling of the entire operating system.

Recalling the past three decades of offensive and defensive confrontation history revolving around control flow, it's not hard to suggest people to regard it as a grand war with a quickly twist. Attackers represented by hackers (of course including White Hats), the originators of the war, continually mine various vulnerabilities such as memory corruption, use-after-free, memory disclosure by all means and tirelessly construct input to trigger them and utilize any accessible services to compromise the defenses in operating system. They brought a boom of new tools and weapons which push forward the war's development. From the other side, all kinds of technologies has also been applied by security engineers, such as malicious input pattern recognition [5], control flow analysis [6], heterogeneous ISA [7], hardware virtualization [8] and so on. They continually

come up with new countermeasures which not only change the modern system's features but also promote these technologies' inward development.

This paper will present all these attack technologies and their corresponding mitigation defenses in metaphor for a war. We will elaborate the causes of their emergence, the principle of their implement and the security characteristics of their method. The paper's main part is divided into three logic chapters. The first chapter is the "Curtain" introducing Code Injection Attacks and Data Execution Protection; the second chapter is the "Upgrade" introducing Code Reuse Attack and Control Flow Integrity; the third chapter is the "Climax" introducing Code Information Disclosure and Randomization. The last part of the paper is "Summary" which look back and rethink the entire field.

Code Injection Attack

The so-called code injection attack, because the software input data to the user is not correct (usually because the software designer does not consider some special type or length of data), so that these data can executable permissions, and the attacker to use, in the data Injection of attack code, arbitrary code execution to achieve the means of attack. The malicious code injected by an attacker is often called ShellCode.

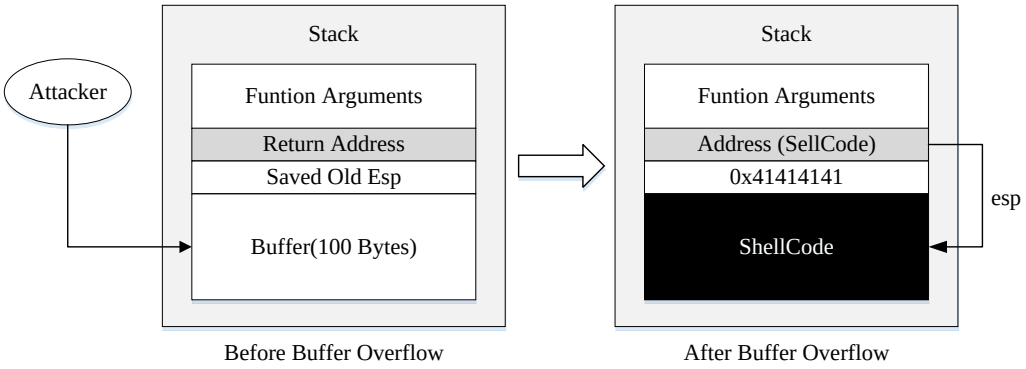

Figure 1. Memory layout of Code Injection Attack

Stack-Smashing. Although code injection can lead to malicious execution, not all injection attacks can achieve complete control flow hijacking, such as the common SQL injection [9], XML injection [10]. Nevertheless, a notorious attack in the field of software security, the Stack-Smashing Attack [11], can achieve full control flow hijacking. The method behind Stack-Smashing Attack is very simple, that is there is a over-bound memory write vulnerability existing in the victim program or its dependent library, which make the vulnerable function's stack frame can be rewrote by the user input data, which was the address of the ShellCode injected by the attacker, so that the execution of return instruction will cause the function return to the malicious ShellCode.

Implementing of Stack-Smashing Attack rely on buffer overflow vulnerabilities in victim program which however are common bugs in large software developed by low-level programming languages which has not automatic pointer bound checking, such as C. Therefore from the early 90s of last century so far, the Stack-Smashing Attack has been one of the most serious problems in the security field, plagued the majority of developers and systems engineers. Famous Worm Attacks and Botnet Attacks causing great harm to the computer world, both can be achieved by Stack-Smashing control flow hijacking. And then, the Heap-Smashing alike Attack with similar simple principle but more complex structure, such as Heap Spray [12] and Heap Fengshui [13], is presented. The method of control flow hijacking is to rewrite the object's virtual table pointer (VPTR) saved on the heap by buffer overflow which make it point to the fake virtual tables including function pointers to the ShellCode constructed by attackers. Fig. 1 shows the mechanism of Smashing-Stack Attack.

Bound Check. As far as above description, the root reason of Smashing Attack is over-bound access, so a group of program analysis experts proposed to the Bound Check defense. This

mechanism has two implementations: one is find all memory access points and its range, then determine whether it's legal in the current procedural context through program slicing technology and abstract interpretation during compilation time; the other is instrument in program's statements at compile time which can not only record and update the scope of pointers but also can trace the generation and propagation of the involved data. Furthermore, the stub functions are also inserted before all access statements which can check the bound before visiting. From the perspective of compilation technology, the first method belongs to the static method (check at compilation), the second belongs to the dynamic method (check at program running).

As with other program analysis methods, they have inherent disadvantages: static methods cannot guarantee completeness and soundness because they rely on point-to analysis, which often results in false positives and false negatives; dynamic methods need to insert a lot of code which will bring a lot of running overhead.

Later, some scholars have put forward an improvement method, that is, design a new self-access-boundary-check language on the basis of existing language, such as Ccured [14], a C language variant. But the biggest drawback of this approach is not forward compatible with the lots of legacy code in real world, so it is difficult to be accepted by the mainstream industry. And then, scholars have proposed a simplified mechanism of dynamic methods, namely, SFI [15]. SFI discarded the insertion and deletion of pointers, so the overhead can be reduced. However, the performance of the check instrumentation cannot be eliminated, so the performance also cannot be accepted by industry. After that, the DFI mechanism [16], which matches the relationship between the value and the reference of dynamic address check, has been presented, but it has not been accepted by the majority of developers too.

Data Execution Prevention. In the face of the huge threat of code injection attack, the system engineer presents a simple and effective solution from another point of view, the Data Execution Prevent (DEP). The starting point of DEP is very simple, that is to ensure that the data area of program must not have executable permissions (data area refers to the heap area, stack area and global data area). To achieve this target can leverage the page table access control mechanism within most of operating systems. Because the DEP mechanism is easy to implement with little overhead, and can gain obvious defense effect, it has been deployed among a variety of commercial operating system (Windows from XP SP3, Linux kernel from 2.4.2 both adopted this technology). This technique is also named as NX or W ⊕ X.

Sandbox. Of course, in some special cases, the user input data must have the permission to execution, such as the JIT compiler which will read the script's source code as data in the runtime and step by step to compile the code data to executable instructions. For example, most modern browsers' kernel often integrate JavaScript and ActionScript JIT compiler in order to support dynamic Web pages and Flash applications. In this scenario, if attackers construct malicious Web pages to conduct JIT compiler to generate malicious instructions, the purpose of code injection and control flow hijacking can be achieved too [17].

To solve this problem, system engineers have proposed sandbox technology [18]. Sandbox is a lightweight process-level virtual machine, the principle behind is simulating JIT compiler's function by dynamic binary translation [19], and restrict simulated instructions' access among code cache and data buffer, if once they need to access external data or code, they must invoke some library functions wrapper by sandbox which add Parameters Sanitizer to ensure that the program will not perform malicious acts. Sandbox technology has been widely used in a variety of popular browser kernel, such as IE, chrome, Safari, Firefox and so on.

Code Reuse Attack

After the popularity of DEP technology, the prevalent of code injection has been effectively curbed. But then a more sophisticated and powerful attack technology, Code Reuse Attack make the situation of the war between Control Flow Protection changes again. Instead directly inject ShellCode into program's memory space like Stack-Smashing Attack, Code Reuse Attack just

injects some code pointers which pointed to the origin code of the victim program. Once program dereferences these pointers, the control flow is hijacked to the code. Like Stack-Smashing Attack, Code Reuse Attack also need to rewrite return address on the stack or some other code pointers by some buffer overflow vulnerabilities. When comes to this, you must doubt if original benign code can do evil actions? The answer is yes, for example some standard library functions, such as the function “system”, the attacker can upload a malicious Shell script and execution it by this function, and this script may be a virus or a Rootkit. The attack utilizing standard library functions to also known as return-to-libc [20]. Because code reuse attacks do not directly upload malicious instructions, there is no violation of DEP mechanism. Additionally implementation is not very complicated, so the emergence of this attack became a rapid inundation.

Return-Oriented Programming. With the development of Code Reuse Attack, the granularity of reused code is more subtle and its function is more powerful, until 2007, the famous Return-Oriented Programming (ROP) model was proposed. ROP model’s reusing granularity is no longer a whole function, but some instruction fragments which all ends with a “ret” instruction. These fragments are called "Gadget". Because the tails are all “ret” instructions, when executes to the end, the Gadget will fetch its return address pointed to next Gadget saved on the stack which was already constructed by the attacker through buffer overflow. By this way, the attacker can use some Gadgets to achieve his Attack logic. These Gadget addresses injected on stack, are usually called payload. It has been proved that the ROP model is Turing Complete, that is, in theory, ROP can simulate any Turing Machine computable function [21]. Fig. 2 is the runtime principle of ROP attack.

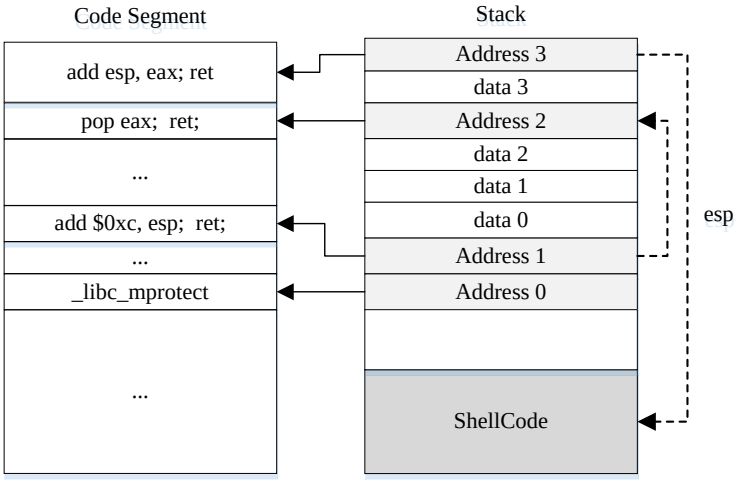

Figure 2. The Stack layout of ROP Payload

Automatic Exploit and Defense. Although the ROP attack function is more powerful, but the artifacts of ROP payload is very difficult. This requires for the attackers not only a lot of experience in the development of assembly language programs, but also a lot of time to filter, construct and debug Gadget chain. In addition, in the actual attack scenario, after the injection of payload through buffer overflow, it is often necessary to execute some original instructions before return instruction or other control flow hijacking point. These instructions may have some side effects such as access some data by the pointer saved between overflowed buffer and return address, because these pointers have been destroyed by attacker, there operations will cause the victim program crash. This problem also exists in other buffer overflow attacks. Furthermore, some compilers will insert canaries before return address in the program [22], the canaries’ completeness before “ret” execution. So in order to solve these problems, attackers began to ask for the help of Automation technology.

In the era of information booming, Automation technology has become an indispensable part in the field of offensive and defensive confrontation. The traditional security techniques tend to intersect with other technologies such as software engineering, hardware design, and artificial

intelligence. For example, [23] proposed vulnerability pattern recognition technology which can help attackers quickly locate target vulnerabilities through results learning from a large number of vulnerability instances. And also, [24] proposed a method which can intelligently construct attack input data and ROP payload according to different vulnerabilities by formal semantic induce technology. What's more, many of the intrusion detection system and anti-virus software has used machine learning technology to identify malicious code characteristics for a long time. Technology integration in software security especially in the field of control flow protection has already been inevitable wave from now on.

Control Flow Integrity. Shortly after Code Reuse Attack arising, a new defense, "Control Flow Integrity (CFI)", was proposed [25]. The principle of CFI is to check whether the transfer target is legitimate before the occurrence of control flow transfer by instrument. Control flow Transfer usually refers to indirect jump instructions, indirectly call instructions, return instructions at the level of binary, and legitimate targets is usually calculated by Static Control Flow Analysis, or by Dynamic Trace (the legal return address can be recorded in shadow stack during the runtime). CFI regresses the essence of control flow protection which can prevent any attack hijacking the program's control flow by modifying code pointers (indirect transfer target).

However, as with the Bound Check method mentioned above, this method which relies on program statically analysis and instrument, deserves careful consideration both in terms of security and overhead. First, for Static Control Flow Analysis, the accuracy of the point-to analysis affects the final set of transfer targets, however in order to avoid false positives to cause program crashing, the solution set is usually complete but not sound, so a large number of fake targets are included, and these targets are likely to be exploited by attackers. What's worse, the absence of source code which make the analysis can only rely disassemble technology will lead to a more serious loss of analysis accuracy. Second, runtime overhead for stubbed code is a big issue. Like "ret" instructions in functions occur frequently which will make the overhead of the trace and check code reach about 30% [24]. These problems greatly limits the usefulness of CFI.

Therefore, security engineers or developers of practical applications often take a compromise, which sacrifice accuracy and security in exchange for performance improvement. [26] proposed a coarse-grained CFI implementation method for the raw binary program, [27] solved the problem of adding CFI check on dynamic loading library, [28] proposed a path-sensitive coarse-grained CFI method, and so on. With the unremitting efforts of these researchers, CFI technology has finally been accepted by the industry. 2013, Microsoft released her Win8.1 system which deploys coarse-grained CFI technology in EMET4.1.

Address Space Layout Randomization

Compared to injection attacks, code reuse attacks require an important condition, the address of the reuse code before attack. At the beginning of popularity of Code Reuse Attack, most programs and shared libraries were compiled and linked according a fixed absolute address that is base address of code segment when loaded by OS, so once know the version of target software, attacker also know the address and then reuse any code of it. In order to solve this problem, compiler designers have proposed a position-independent code (PIC) compilation technology, so that the program can be loaded at any address in memory, and system engineers also change the OS's loader which will load the PIC code randomly at arbitrary addresses in virtual memory space, thus undermining the basic condition of Code Reuse. This defense mechanism is called Address Space Layout Randomization (ASLR). Because having a small overhead but gaining fair well defense effect, the ASLR has been accepted and deployed by most commercial OS vendors.

Code Information Disclosure. In order to break through the limitation of ASLR, attackers try to leverage Memory Disclosure vulnerabilities to leak the base address of code segment or some important library functions. In general, the Imported Address Table (IAT) in PE format executable programs, the Procedure Linkage Table (PLT) in ELF format executable programs, and the ".rodata" section holding virtual function tables of C ++ programs all contain pointers to code segment which can be exploited by attackers to crack ASLR [29].

Code Diversify. In order to mitigate disclosure vulnerabilities of code information, security experts have proposed an update strategy, fine-grained Randomization or rather called Code Diversify. The original ASLR only randomize the loading base address of code segment, which is easily cracked by deducing some functions' offset, however the fine-grained Randomization changes the order of inner instruction of code segment, so that even if attackers get the base address, they cannot infer all functions and instructions' address, and thus cannot complete malicious behavior. Of course, changing the order of instruction storing must ensure that the topology order of execution is maintained or the results are unchanged. [30] presented the basic-block granularity randomization, [31] presented the basic-block's internal instruction granularity randomization.

JIT-ROP. Although Code Diversify (or fine-grained Randomization) makes it harder to implement Code Reuse Attacks, it does not address the fundamental problem of Code Information Disclosure. So the JIT-ROP technology proposed in 2013 completely nullifies all of them. This time attackers exploit the Code Disclosure vulnerabilities repeatedly to analyze the transfer instructions' offset from an initial leaked code snippet to find other code snippets, until finally collecting enough code fragments to complete the attack [32]. Fig. 3 shows the mechanism of JIT-ROP.

After a deep rethought on randomization mechanism, security exports proposed two defense mechanisms: Lifetime Randomization [33] and Code Pointer Integrity (CPI) [34]. The so-called Lifetime Randomization, is based on the fine-grained Randomization, additionally the running process can continuously change the order of inner instructions in its memory space which to ensure that the leaked code location information is invalid at the time of Control Flow Hijacking. Although this method can effectively defend against JIT-ROP attack, there are still some problems in its scalability and performance and it's not very mature. The Code Pointer Integrity, that is, all code pointers in memory space must be isolated or encrypted, when the program need to dereference these pointers, it must use the private key saved in a private register to decrypt, which to ensure that the code pointer information is not compromised. Compared to Lifetime Randomization mechanism, CPI is more practical and more effective. In the future, it's very likely to be accepted and deployed by OS vendors and compiler developers.

Conclusion

From Code Inject Attack to Code Reuse Attack, and then to Code Information Disclosure, this war around Control Flow Protection is still in full swing, and a variety of ever-changing attack and defense technology also emerge in an endless stream, however we can still find some specific laws and guidelines in software security from looking back at the history of the past 30 years. These rules and guidelines can not only provide a scientific and effective thinking method for the security field engineers when design a new technology, but also can offer some suggests for all level developers to evaluate their products' safety and performance. The following three points that we summarize are the most important.

First. The target semantics and underlying implementation of the software have differences which are likely to be exploited by attackers. The above mentioned Buffer Overflow (BoF), Memory Disclosure, and common Use-After-Free (UAF) are all due to the fact that the developer does not take into account that the programming language does not handle the special input case when implementing the target semantics, resulting in an exceptional execution flow (or a undefined behavior) which lead to Control Flow Hijacking ultimately. The defender in the design of the appropriate means of defense, but also need from this point of view, first construct the attack vector, that is, the conditions for the implementation of dependency attack, but for one of the conditions, the design of defense mechanisms.

Second. Absolute security system does not exist and the superiority of the software security field take turns. This point for the defenders is, do not over-pursuit of security while ignoring other considerations. A defense mechanism accepted by developers and system vendors must keep the balance of security, performance, scalability and compatibility. The CFI mechanism mentioned above is the best example.

Third. The actual attack and defense will involve a large number of other areas of technology. As a comprehensive hacker or a qualified security engineer, you must have a in-depth understanding knowledge on the programming language, computer architecture and operating system-related. What's more, you should be able to utilize data modeling and data analysis algorithms proficiently, and be aware of software engineering and artificial intelligence-related methods. Software security, has the the same direct future as other fields of information technology, that is specific, digitization and intelligent.

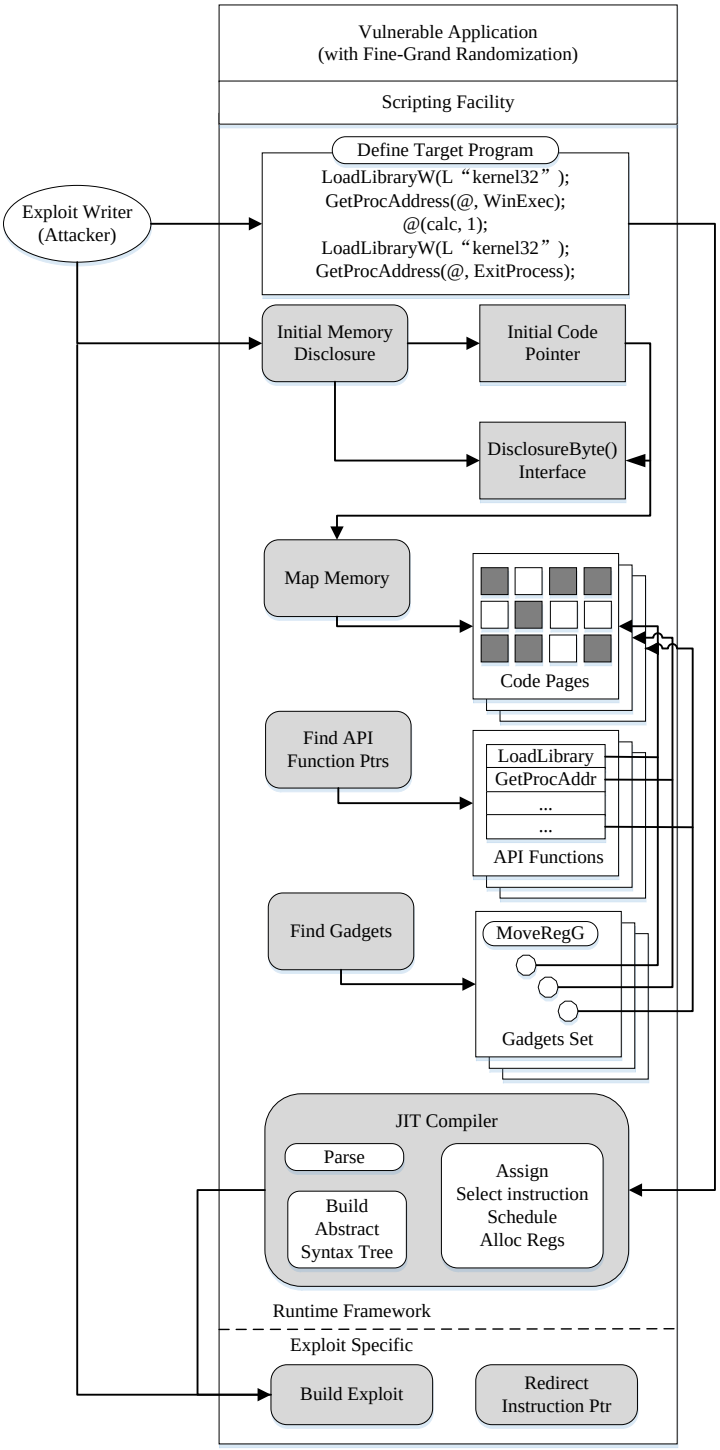

Figure 3. Overall workflow of JIT-ROP

Acknowledgement

We would like to thank anonymous reviewers for their useful feedback. This research is supported by the Beijing Municipal Science & Technology Commission Program under grant D161100001216002.

References

[1] Tyrone Grandison, Morris Sloman. A survey of trust in internet applications[J]. Journal of IEEE Communications Surveys & Tutorials. 2000, 3(4): 2-16.

[2] Vassilios S. Verykios, Elisa Bertino, Igor Nai Fovino, Loredana Parasiliti, Yucel Saygin, Yannis Theodoridis. State-of-the-art in privacy preserving data mining[J]. Journal of ACM SIGMOD Record. 2004, 33(1): 50-57.

[3] S.R. Subramanya, B.K. Yi. Digital rights management[J]. Journal of IEEE Potentials, 2006, 25(2): 31-34.

[4] R.S. Sandhu, P. Samarati. Access control: principle and practice[J]. IEEE Communications Magazine, 2002, 32(9): 40-48.

[5] M. Christodorescu, S. Joa. Static Analysis of Executables to Detect Malicious Patterns[C]. Proceedings of the 12th conference on USENIX Security Symposium. 2003, 12: 12-12

[6] F. Nielson, Hanne Riis Nielson, Chris Hankin. Principles of Program Analysis[M]. New York City, USA: Springer. 2005.

[7] A. Venkat, S. Shamasunder, H. Shacham. HIPStR: Heterogeneous-ISA Program State Relocation[C]. Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems. USA. 2016: 727-741.

[8] Monirul I. Sharif, Wenke Lee, Weidong Cui, Andrea Lanzi. Secure in-VM monitoring using hardware virtualization[C]. Proceedings of the 16th ACM conference on Computer and communications security. USA. 2009: 477-487.

[9] W.G. Halfond, Jeremy Viegas, Alessandro Orso. A Classification of SQL-Injection Attacks and Countermeasures. Proceedings of the International Symposium on Secure Software Engineering. USA. 2006: 325-337.

[10] Thiago Mattos Rosa, Altair Olivo Santin, Andreia Malucelli. Mitigating XML Injection 0-Day Attacks through Strategy-Based Detection Systems[J]. 2012, 11(4): 46-53.

[11] Jonathan Pincus, Brandon Baker. Beyond stack smashing: recent advances in exploiting buffer overruns[J]. IEEE Security & Privacy. 2004, 2(4): 20-27.

[12] Mark Daniel, Jake Honoroff, Charlie Miller. Independent Security Evaluators Engineering Heap Overflow Exploits with JavaScript[C]. Proceedings of the 2nd conference on USENIX Workshop on offensive technologies. USA. 2008.

[13] Gene Novark, Emery D. Berger. DieHarder: securing the heap[C]. Proceedings of the 17th ACM conference on Computer and communications security. USA. 2010: 573-584.

[14] George C. Necula, Scott McPeak, Westley Weimer. CCured: type-safe retrofitting of legacy code[J]. ACM SIGPLAN Notices - Supplemental issue. 2012, 47(4): 74-85.

[15] Robert Wahbe, Steven Lucco, Thomas E. Anderson, Susan L. Graham. Efficient software-based fault isolation[C]. Proceedings of the fourteenth ACM symposium on Operating systems principles. USA. 1993: 203-216.

[16] Miguel Castro, Manuel Costa, Tim Harris. Securing software by enforcing data-flow integrity[C]. Proceedings of the 7th symposium on Operating systems design and implementation. USA. 2006: 147-160.

[17] Ping Chen, Yi Fang, Bing Mao, Li Xie. JITDefender: A Defense against JIT Spraying Attacks[C]. IFIP International Information Security Conference. USA. 2011: 142-153

[18] Bennet Yee, David Sehr, Gregory Dardyk. Native Client: A Sandbox for Portable, Untrusted x86 Native Code[C]. 30th IEEE Symposium on Security and Privacy. 2009.

[19] K. Ebcioglu, E. Altman, M. Gschwind. Dynamic binary translation and optimization[J]. IEEE Transactions on Computers. 2001, 50(6): 529-548

[20] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)[C]. Proceedings of the 14th ACM conference on Computer and communications security. USA. 2007: 552-561

[21] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko. Return-oriented programming without returns[C]. Proceedings of the 17th ACM conference on Computer and communications security. USA. 2010: 559-572.

[22] William H. Hawkins, Jason D. Hiser, Jack W. Davidson. Dynamic Canary Randomization for Improved Software Security[C]. Proceedings of the 11th Annual Cyber and Information Security Research Conference. USA. 2016.

[23] Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey[J]. Journal of Information Security and Applications. 2009, 14(1): 16-29.

[24] Edward J. Schwartz, Thanassis Avgerinos, David Brumley. Q: exploit hardening made easy[C]. Proceedings of the 20th USENIX conference on Security. USA. 2011: 25-25.

[25] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, Jay Ligatti. Control-flow integrity[C]. Proceedings of the 12th ACM conference on Computer and communications security. USA. 2005: 340-353.

[26] Mingwei Zhang, R. Sekar. Control flow integrity for COTS binaries[C]. Proceedings of the 22nd USENIX conference on security. USA. 2013: 337-352.

[27] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, Cristiano Giuffrida. Practical Context-Sensitive CFI[C]. In Proceedings of the 22nd ACM Conference on Computer and Communications Security. USA. 2015: 927-940.

[28] Ben Niu, Gang Tan. Modular control-flow integrity[C]. Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. USA. 2014: 577-587.

[29] Mark Dowd, Alexander Sotirov. Bypassing browser memory protections in Windows Vista[C]. Proceedings of Black Hat USA Conference. USA. 2008.

[30] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, Zhiqiang Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code[C]. Proceedings of the 2012 ACM conference on Computer and communications security. USA. 2012: 157-168.

[31] Georgios Portokalidis, Angelos D. Keromytis. Fast and practical instruction-set randomization for commodity systems[C]. Proceedings of the 26th Annual Computer Security Applications Conference. USA. 2010: 41-48.

[32] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization[C]. Proceedings of the 2013 IEEE Symposium on Security and Privacy. USA. 2013: 574-588.

[33] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, Lexington, Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures[C]. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. USA. 2015: 268-279

[34] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, Dawn Song. Code-pointer integrity[C]. Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation. USA. 2014: 147-163