Runtime Code Reuse Attacks: A Dynamic Framework Bypassing Fine-Grained Address Space Layout Randomization

Yi Zhuang[1], Tao Zheng[12], Zhitian Lin[1]
Software Institute[1]. National Key Laboratory for Novel Software Technology[2]
Nanjing University
Nanjing, China
{mg1132019, zt, mg1232007}@software.nju.edu.cn

***Abstract*—Fine-grained address space layout randomization has recently been proposed as a method of efficiently mitigating ROP attacks. In this paper, we introduce a design and implementation of a framework based on a runtime strategy that undermines the benefits of fine-grained ASLR. Specifically, we abuse a memory disclosure to map an application's memory layout on-the-fly, dynamically discover gadgets and construct the desired exploit payload, and finish our goals by using virtual function call mechanism—all with a script environment at the time an exploit is launched. We demonstrate the effectiveness of our framework by using it in conjunction with a real-world exploit against Internet Explorer and other applications protected by fine-grained ASLR. Moreover, we provide evaluations that demonstrate the practicality of run-time code reuse attacks. Our work shows that such a framework is effective and fine-grained ASLR may not be as promising as first thought.**

Keywords-code reuse; security; dynamic; fine-grained ASLR

I. INTRODUCTION

Software vulnerabilities have been a major cause of computer security incidents. Buffer overflows [3, 5], integer overflows and heap overflows were used to pose a significant threat to modern operating systems [1]. Format string vulnerabilities [2] allow an attacker to control the first parameter to a function of the printf-family, which can be used to store pointers to specific addresses if it is placed on the stack. Despite differences in the style and implementation of these exploits, they all share a same goal: to achieve the control-flow hijacking attempts within the vulnerable application. Nowadays, numerous defenses have been implemented to limit the scope of these attacks. However, well motivated attackers still succeed in their intent. So the cat and mouse game plays on.

To thwart such attacks, many mitigation techniques have been developed. Address Space Layout Randomization (ASLR) [9] and Data Execution Prevention (DEP) are very real thorns in the side of an attacker. DEP makes locating shellcode difficult; the attacker must find a page with executable permission and find a way to write to it when it has writable permission and figure out the location. Attackers then redirect to code reuse attacks. This new strategy utilizes code already present in memory, instead of relying on code injection. The canonical example is return-to-libc [6, 8], in which exploits redirect control-flow to existing shared-library functions. But if the base address of the memory segment is randomized, then the success rate of such an attack significantly decreases. Shacham [7] introduced a new approach named return-oriented programming, which chains together short instruction sequences ending with a ret instruction (called gadgets) that already exists in the memory of the application and executes some specific computation. The key idea of ASLR is to randomize the base address of the stack, heap, code, and dynamic libraries at load and link time, which offered a plausible defensive strategy against these attacks. But a drawback of this approach is that not all memory regions have been protected with ASLR, the address space for 32bit binaries is small which opens the possibility of probabilistic attacks [17]. Besides that, ASLR on 32-bit architectures only leaves 16 bit of randomness, an attacker might attempt to perform a brute-force attack.

After that, smart defenders have been busily working to fortify perimeters by designing fine-grained randomization strategies [21] for repelling the next generation of wily hackers. Some approaches introduce randomness at compile time. For example, compilers can be modified to generate code without ret instructions [25]. But these mechanisms fail to handle attacks leveraging jmp instructions. Marlin, introduced by Gupta [18], randomizes the function-level structure of the executable code, so denying attacker the necessary a priori knowledge of instruction addresses for constructing such a desired exploit payload. Other approaches have also been proposed to randomize processes. STIR [19] and XIFER [26] defend against ROP by randomizing at the basic block granularity. ILR [24] randomizes the location of each instruction in the virtual address space and uses a process-level virtual machine to find the called code, which imposes a significant on-going performance cost. Therefore, the attacker is unaware of how exactly permutation is randomized for the currently executing process image. So the radiance of traditional ROP attack is fading away. We define these mitigations of ROP embodied by fine-grained ASLR.

In this paper, through memory disclosure, we implement a completely new variant of code reuse attack wherein we gather code chunks and retrieve them to the desired payload dynamically from memory layout during the vulnerable application is running. Then treating the calling of virtual

This work is funded by National Natural Science Foundation of China under Grant No.61373010

function as the trampoline, replace the address of function in vtable with gadget's first addresses in proper sequence. Finally trigger the call of function to complete our ROP attack on-the-fly. We show strong evidence that our variant ROP attack can entirely bypass all fine-grained randomization scheme and ROP mitigation. Based on the above findings, we argue that the fine-grained ASLR strategies still have loopholes. Meanwhile, we hope that our work will inspire others to explore more comprehensive defensive strategy than what exists today.

II. BACKGROUND

We review the necessary technical background information before introducing the methodology behind our attack.

A. Code Reuse Attacks

The fundamental factor for code reuse attack is that the relative offsets of instructions in the application's code are constant. That is to say, if an adversary knows any symbol's address in the application code, then the location of all gadgets and symbols in application's code is deterministic.

Return Oriented Programming [14] is generalization of return-to-libc attack [6, 8], which involves an adversary redirecting the program execution to an existing library function [12]. The general principle of any ROP attack is to combine short instruction sequences found in memory (or whatever code is not randomized), called gadget, and allowing an adversary to perform arbitrary computation. Recently, this concept was overthrown by removing the reliance on return instructions [13]. However, we show the basic idea of code reuse using ROP for simplicity in Figure 1. Steps ① to ⑦ show the entire procedure of ROP attacks.

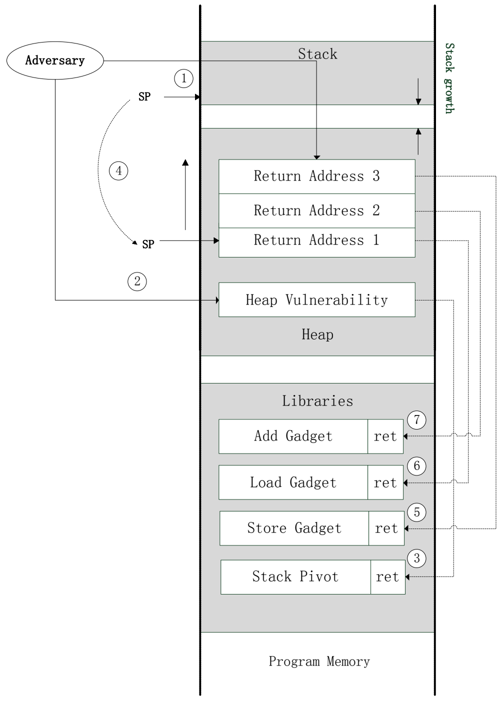

Figure 1. Layout of a sample ROP attack on the heap using a sequence of single-instruction gadgets.

Jump Oriented Programming (JOP) [10, 29] is similar to ROP in that JOP manipulates the control flow of the application. Jump oriented data is not limited to stack overflows but uses modified indirect control flow transfers to construct the chain of executed gadgets. Indirect control flow transfers are used in the application to support, e.g., library calls, function pointers, and object oriented programming. JOP has similar limitations like ROP. In addition, JOP needs to redirect control flow to the first JOP dispatcher. ASLR severely limits the initial redirection for JOP.

B. Fine-Grained Randomization for Exploit Mitigation

A widely accepted countermeasure against code reuse attacks is that the defender randomizes the application's memory layout. This scheme randomizes the base address of segments such as the stack, heap, shared libraries, and the executable code itself. As is shown in Figure 2, the start address of an executable is changed and sequence of gadgets is shuffled between consecutive runs of the same application, which is not intended by the attacker. So the adversary must guess the location of the instruction sequences needed for deployment of their code reuse attack.

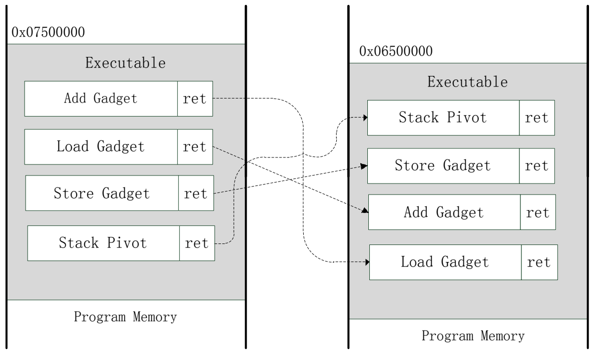

Figure 2. Mitigation of ROP attack by fine-grained memory and code randomization

Unfortunately, current ASLR implementations only randomize on a per-module level, which suffer from two main problems: first, ASLR can be bypassed by means of brute force attacks [15, 17] because the entropy on 32-bit systems is too low. Furthermore, upgrading to 64-bit is simply not feasible. Second, all ASLR solutions are vulnerable to memory disclosure attacks [11, 16] where the attacker gains knowledge of a single runtime address and uses it to re-enable code reuse. Therefore, some gadgets from a disclosed region can be organized deliberately by the adversary offline before deploying the exploit. To confound these attacks, Industrious defenders put forward a number of fine-grained ASLR and code randomization schemes [18, 19]. The broadly idea in these works is to randomize the data and code structure, for instance, by shuffling address of functions or basic blocks. Figure 2 depicts that the performance of these approaches are that the location of all gadgets is randomized.

We assume target platform uses the following mechanisms to mitigate the execution of malicious computations:

- Non-Executable Memory: We assume that the target vulnerable application is under the protection of non-executable memory (also called NX or DEP) applied to the heap and the stack. It will make the traditional code-injection attack ineffective. We also assume that this mechanism prevents adversary from tampering all executables and native libraries, in order to stop one from overwriting existing code.

- Base Address Randomization: We assume that the target platform randomizes base addresses of library and executable segments effectively. Mappings have been eliminated.

- Fine-Grained ASLR: We assume that the target platform deploys strong fine-grained memory and code randomization scheme on executables and libraries. First, target platform shuffles the order of functions [18] and basic blocks [19]. Second, target platform swaps registers and replaces instructions [20]. Third, target platform randomizes the location of each instruction [23] and performs randomization upon each execution of vulnerable application [19].

III. OVERVIEW OF RUNTIME CODE REUSE ATTACK

Snow [30] has put forward the concept of runtime code reuse and implemented a framework that can launch exploit on-the-fly. But the drawback in this framework is very obvious. First, Snow did not give out the detail about how to obtain the entry point of memory disclosure. Second, Just-in-time compilation is simply too heavy-weight to be consider in a runtime framework. In our framework, we explain how to disclose code segment in detail, propose a new method to chain every gadget which getting rid of dependence on stack and proved to be light-weight.

The overall work flow of our runtime exploit is shown in Figure 3. In Step ①, we arrange objects on the heap in order by defragmenting system heap. Next, we perform an overflow on JS object to disclose the memory and construct our code chunks when target vulnerable application is running. Then we pick up some useful gadget sets from code pages. In Step ②, through setting up fake objects on the heap, we embedded serialized code chunks in our vtable. Through utilizing mechanism of virtual function call in target program, unordered gadget sets become malicious ROP payload. At last, we inject all above information in our exploit script to trigger our malicious attack in Step③.

A. Disclosing Memory

Because we assume our target application is running on the most fine-grained platforms, so memory layout of application is totally different at each execution. Therefore, the attempt to apply static code analysis is useless and futureless. The entire procedure of our attack is happening during the vulnerable application is executing. We need to arrange objects on the heap in the following order: (i) buffer to overflow, (ii) string object, (iii) JS object. The state of a process's heap depends on the history of allocations and deallocations that occurred during the process's lifetime. Unfortunately, we do not know the heap state resulting from a specific sequence of previous allocations.

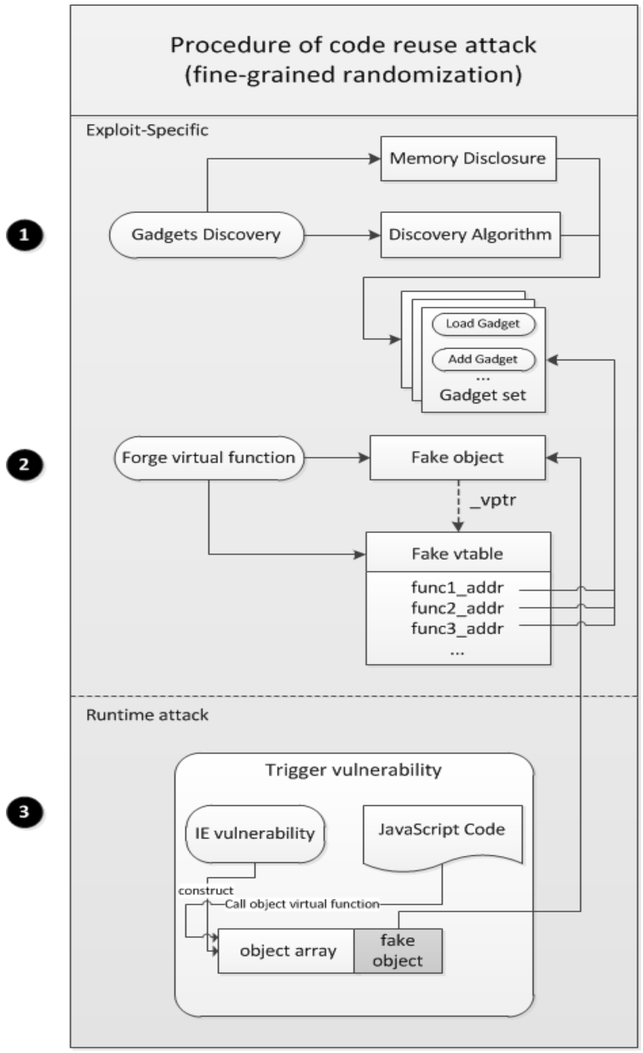

Figure 3. Workflow of memory code reuse attack against a script-enabled application protected by fine-grained ASLR

We can image that there are many holes of free memory. To overcome this hurdle, we must defragment the heap. Our purpose is to fill up these "holes" which are bigger than we allocate later on. This can be done by allocating a large number of blocks of the size we will use. After that, every time we allocate new blocks from the end of the heap. Thus the behavior of the allocator will be equivalent to starting with an empty heap.

One important exception is the data for JavaScript string object which is stored as BSTR [27] used by the COM interface. BSTR strings are stored in memory as a structure containing a four-byte size field, followed by the string data and a two-byte null terminator. The first block is the following string object. We take advantage of unsigned int overflow, so that actually used size in block is smaller than we allocated earlier. Then we fill the last four-byte with "0xffffffff". So the following string object's first four-byte is overflowed and string length will cover entire 32-bit address space layout. This will allow us to read bytes at any relative address in memory. As the relative offset from string object to button object is known, we can use the relative read to reveal the absolute address through self-reference first 4-byte field in third block. We implement a method that, given a relative address, switches it to an absolute address and reveals the corresponding data. In short as memory is discovered, there will be many code chunks in code pages for us to build a useful payload, which we elaborate on in the next sections.

B. *Gadget Discovery*

Now our task lies in finding out a number of gadgets to use as our code reuse payload. Not every instruction sequence can be used as a gadget. We choose our usable gadgets in the following three steps:

1) Upon with the code pages dynamically discovered in process's memory, we generate the sequence of instruction gadgets ended with ret by adapting the Galileo [4] algorithm and record them in a trie.

2) Discover the gadget that has the same semantic definitions [28] with our desired gadget of payload in the trie. If not found, we disclose the memory to glean new code pages and return to the first step.

3) We filter the gadgets discovered in step two using a kind of instruction dependence method which will be illustrated later. If no gadgets left after the filtration, we return to the first step.

Since fine-grained ASLR mitigation may change the module of code pages on each execution, we could not have much energy to analyze the code chunks offline. What we could do is dynamically collect gadgets at the time exploit runs. Furthermore, we must do this work as quickly as possible because our attack must run in real-time before the victim machine terminates the vulnerable application. Unfettered access to extensive memory address space enables us to search for gadgets. We iterates over the code pages to collect useful gadgets by adapting the Galileo [4] algorithm. After that we use semantic definitions to match gadgets and introduce a kind of instruction dependence method to filter them. In terms of the instruction dependence method, we define the dependence (Ins_i, Ins_j) as following.

- If there exists a register Reg_a which serves as a destination register in Ins_m and as a source register in Ins_{m+1}, so the execution of Ins_m depends on Ins_{m+1}, we define Ins_m depends on Ins_{m+1}.
- If Ins_{m+1} is a jump instruction, we define Ins_m depends on Ins_{m+1}. For example, jnz eax, Ins_{m+1} strictly associates with Ins_m, which sets up flag register %eflag.
- If register is involved in accessing memory, that is, Ins_i, Ins_j corresponding to opcode reg_{src}, [eax+Imm_s]; opcode [eax+Imm_t],reg_{des}; if value in the register is not changed and $Imm_s = Imm_t$, we define Ins_i depends on Ins_j; if value is changed, we define Ins_m depends on Ins_{m+1}.
- If Ins_m and Ins_{m+1} correspond to stack operation instruction sequence, such as pop and push instruction, we define Ins_m depends on Ins_{m+1}.

A gadget can be determined as usable or not after we check every adjacent two instructions' dependence according to the above principle. If there are some gadgets left after the three steps, we choose them as our final gadgets and will use their addresses in the runtime attack.

C. *Forging Virtual Function*

The next challenge is how to effectively use the collection of gadgets to satisfy the exploit target program. The traditional sense of the ROP attack lies in a stack-based buffer overflow. A ROP attack constructs a set of stack invocation frames that are popped one after another. Each stack invocation frame prepares a set of parameters on the stack and targets a gadget that uses the parameters and executes some malicious computation. ROP works around DEP but relies on static addresses for the stack and for the gadgets. In addition ROP needs to initially redirect control flow to the first ROP invocation frame.

The core notion of our attack is that we replace function addresses in virtual function table with some gadget's address. When the target application use the associated object, the first virtual function is been called, control-flow is redirected by sequentially executing the gadgets in memory which was found in previous step to fulfill adversary's goal. We do not need to arrange gadgets on stack that ends with ret instruction with the stack pointer, %esp, to execute the next gadget. This method can totally negate the effect of stack protection strategy.

We allocate one blocks of memory on the heap for a fake vtable. In figure 4, fake vtable is an object similar to String object. It has 4-byte head, following by 4-type padding content, next the virtual function addresses, and a 2-byte null terminator. Any virtual function call starts at offset 8 in the vtable. So if we put our gadget's first instruction address in this position, virtual function call through the vtable will jump to a location of our choosing.

Head 4 Bytes	**Paddings 4 Bytes**	**Gadget1_ addr 4 Bytes**	**Gadget2_ addr 4 Bytes**	**Gadget3_ addr 4 Bytes**	**…**	**Paddings**	**Terminator 2 Bytes**

Figure 4. Data structure of fake vtable

Another data structure, called fake object, contains the base address of our fake vtable. This procedure has involved lookaside lists maintained by the system memory allocator. The cache consists with a number of bins, each holding 6 blocks of a certain size range. When a block is freed, it is stored in one of the bins. If the bin is full, the smallest block in the bin is freed and replaced with the new block. In terms of our needs, we apply a certain amount of blocks from memory, which size equals to our vtable, in order to make this size on the lookaside list empty. Now let's free this size of block containing a vtable, so there is a free block whose content is original vtable on the lookaside list and lookaside head will point to this block. We construct the fake object by assigning the lookaside head address to the fake object pointer.

D. *Trigger Vulnerability*

We overwrite the object pointer with lookaside list base address called fake object pointer in Figure 5. Now the fake object pointer points to the virtual function address in vtable that has been overwrote with our first gadget's base address in memory. Every time we call a virtual function, corresponding gadget will be executed. We put a number of object pointers into an array. So if an adversary wants to do high-level malicious computation, he just successively uses these pointers to call virtual functions.

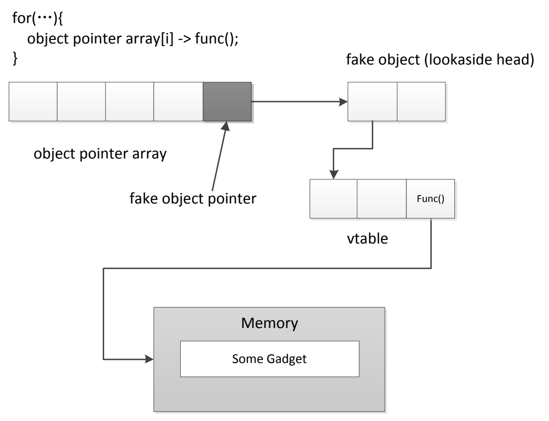

Figure 5. Workflow of virtual function call

Because the times of virtual functional calls determine the scale of our payload constructed by gadgets, if one object is not enough for us, we can construct more objects which include the virtual function calls to increase the power of our attack. After we finish the above steps, what we need is only to invoke vulnerable object which is used by the running application. Finally it will trigger our runtime attack.

E. Implementation

To prove the power of our framework, we use it to exploit Internet Explorer (IE) 10 running on Windows 7 using CVE-2013-2551. The vulnerability is an integer overflow within certain object to achieve code execution in the context of IE 10 sandboxed process. We put our desired objects on the heap, dynamically traverse code pages, semantically pick up gadgets and executing a serialized payload useable through a serious of virtual function calls. All performed at runtime in a script environment.

In Step ①, we develop a JavaScript library called HeapEasy that encapsulate the system allocation operations. For example, the implementation of HeapAlloc interface ensures each blocks allocation comes from the system heap, and implements a HeapLookAside interface to add blocks of the specified size to the lookaside list. Next, iteratively using the charCodeAt() function to read data from the memory code pages and implement a DiscloseAbsAddr interface that translates a relative address to an absolute address. We snapshot the memory state in our hashmap which contains the absolute address of every instruction. At the same time, we dynamically filter the useable gadgets which adapts the criteria from Schwartz [28]. After that, we construct the fake vtable by using HeapVTable interface in Step②.

Generally speaking, a virtual function call needs a MoveRegG to get the vtable address from register ecx, and a JumpG to invoke the function at offset 0x8 in the vtable. Our library enables us to figure out the heap base address and the absolute address of a jmp ecx or equivalent instruction through the debug library called DbgHelp. When IE 10 is running on the target machine which uses the vulnerable ActiveX widget, our script, included in any HTML or other document-format supporting carriers, starts by an amount of bootstrap code. At last, our payload begins execution in Step③.

IV. EVALUATION

We evaluated the proposed attack of our framework by applying it on real-world application under the protection of fine-grained randomization. Overall, the evaluation demonstrated the wide-scale applicability of our runtime code reuse framework against Internet Explorer in Windows and other popular applications.

A. On Runtime Performance

We performed five tests of our framework. The first test uses integer overflow vulnerability in the context of IE10 sandboxed process. The second, third and fourth tests exploit the Internet Explorer plugin on windows. We use a DebugLog function that includes format string vulnerability in the second test. We employed the family of printf functions because of no warnings were reported. In the third test, we took advantage of an arbitrary sequence of commands. So we chose the Windows calculator program. The vulnerability in the fourth test is triggered by creating an ActiveX object and calling its KeyFrame method with an argument overflow larger than 0x07ffffff. The fifth test demonstrates the native performance of our framework.

B. Evaluation of the Defense against Our Framework

Table 2 shows the popular mitigations used in current operation system. First, our framework does not need to arrange each gadget's first instruction address on the stack and overflow the return address in EIP. So GS strategy loses her efficiency. The same principle applies to SafeSEH. Second, DEP strategy introduces a no-execute bit (NX) to prevent stack from code-injection attack. Our framework utilizes code already present in memory, so NX strategy does no harm to our framework. In the academic circles, randomization technique at various levels of granularity –function level, block level and instruction level were put forward recently. However, our framework glean code chunks at the same time , the target vulnerable application was loaded in memory. That is to say, the step of collecting gadgets happens after the application was randomized. The runtime feature of our framework determines the fine-grained ASLR strategy is short-sighted.

Table I RUNTIME FRAMEWORK AGAINST POPULAR ATTACK MITIGATIONS

Attack Mitigations		**Runtime framework**
GS	***Stack cookies***	√
	Variable reordering	√
SafeSEH	***SEH handler validations***	√
	Permanent SEH	√
DEP	***NX***	√
ASLR	***Function-level***	√
	Instruction-level	√

V. CONCLUSION

Fine-grained ASLR has been introduced as an efficient strategy to defense the code reuse attacks like ROP. In this paper, we introduce a dynamical framework that trigger a runtime code reuse by exploiting the virtual function call. Our framework consists of memory disclosure, gadget discovery and fake virtual function call construction. All work is executed after the vulnerable program begins, so the ASLR can hardly affect our experiment. We demonstrate the efficiency and complete some attacks using this framework. This kind of strategy may be a potential signal that future hacker will use the same or similar way to bypass the ASLR and do something dangerous. We hope our work will attract more researchers' attention and inspire them to bring forward more comprehensive defense.

REFERENCES

[1] E. H. Spafford, "The Internet Worm Program: an Analysis," SIGCOMM Computer Communication Review, vol. 19, no. 1, pp. 17–57, 1989.

[2] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In Symposium on Recent Advances in Attacks and Defenses, 2012.

[3] Aleph One. Smashing the stack for fun and profit. Phrack Magazine, 49(14), 1996.

[4] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In USENIX Security Symposium, 1998.

[5] D. Litchfield. Defeating the stack based buffer overflow exploitation prevention mechanism of Microsoft windows 2003 server. In Black Hat Asia, 2003.

[6] Solar Designer. Return-to-libc attack. Bugtraq, 1997.

[7] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In ACM Conf. on Computer and Communications Security, 2007.

[8] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In Proceedings of the Annual Computer Security Applications Conference, pages 60–69, 2009.

[9] ——, "Address space layout randomization." [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[10] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code reuse attack. In ACM Symp. on Info., Computer and Communications Security, 2011.

[11] F. J. Serna. The info leak era on software exploitation. In Black Hat USA, 2012.

[12] Nergal. The advanced return-into-lib(c) exploits: PaX case study. Phrack Magazine, 58(4), 2001.

[13] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In ACM Conf. on Computer and Communications Security, 2010.

[14] D. D. Zovi. Practical return-oriented programming. Invited Talk, RSA Conference, 2010.

[15] L. Liu, J. Han, D. Gao, J. Jing, and D. Zha. Launching return-oriented programming attacks against randomized relocatable executables. In IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 2011.

[16] A. Sotirov and M. Dowd. Bypassing browser memory protections in Windows Vista, 2008.

[17] H. Shacham, E. jin Goh, N. Modadugu, B. Pfaff, and D. Boneh. On the effectiveness of address space randomization. In ACM Conf. on Computer and Communications Security, 2004.

[18] A. Gupta, S. Kerr, M. Kirkpatrick, and E. Bertino. Marlin: A fine grained randomization approach to defend against rop attacks. In J. Lopez, X. Huang, and R. Sandhu, editors, Network and System Security, volume 7873 of Lecture Notes in Computer Science, pages 293–306. Springer Berlin Heidelberg, 2013.

[19] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In ACM Conf. on Computer and Communications Security, 2012.

[20] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In IEEE Symposium on Security and Privacy, 2012.

[21] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. InUSENIX Security Symposium, 2012.

[22] S.Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. http://users.suse.com/~krahmer/no-nx.pdf, 2005.

[23] J. D. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where' d my gadgets go? In IEEE Symposium on Security and Privacy, 2012.

[24] Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: Ilr: Where'd my gadgets go? In: Proc. of the 2012 IEEE Symposium on Security and Privacy, pp. 571–585 (2012)

[25] Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: defeating return-oriented programming through gadget-less binaries. In: Proc. of the 26th Annual Computer Security Applications Conference, pp. 49–58 (2010)

[26] Davi, L., Dmitrienko, A., N¨urnberger, S., Sadeghi, A.R.: Xifer: A software diversity tool against code-reuse attacks. In: 4th ACM International Workshop on Wireless of the Students, by the Students, for the Students, S3 2012 (August 2012)

[27] ——, "BSTR." [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms221069(v=vs.85).aspx

[28] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: exploit hardening made easy. In USENIX Security Symposium, 2011.

[29] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In ASIACCS'11: Proc. 6th ACM Symp. on Information, Computer and Communications Security (2011), pp. 30‑40.

[30] Snow, Kevin Z., et al. "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization." *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013.