The ISC Int'l Journal of Information Security

ISeCure

January 2017, Volume 9, Number 1 (pp. 53–72)

http://www.isecure-journal.org

Dwarf Frankenstein is Still in Your Memory: Tiny Code Reuse Attacks

AliAkbar Sadeghi [1,*] Farzane Aminmansour [1], and HamidReza Shahriari [1]

[1] *Department of Computer Engineering and Information Technology, Amirkabir University of Technology, Tehran, Iran*

ARTICLE INFO.

Article history:
Received: 2 December 2015
First Revised: 3 January 2017
Last Revised: 24 January 2017
Accepted: 25 January 2017
Published Online: 30 January 2017

Keywords:
Software Security, Code Reuse Attacks, Jump Oriented Programming, TinyJOP, Kernel Trapper Gadget.

ABSTRACT

Code reuse attacks such as return oriented programming and jump oriented programming are the most popular exploitation methods among attackers. A large number of practical and non-practical defenses are proposed that differ in their overhead, the source code requirement, detection rate and implementation dependencies. However, a usual aspect among these methods is consideration of the common behaviour of code reuse attacks, which is the construction of a gadget chain. Therefore, the implication of a gadget and the minimum size of an attack chain are a matter of controversy. Conservative or relaxed thresholds may cause false positive and false negative alarms, respectively. The main contribution of this paper is to provide a tricky aspect of code reuse techniques, called tiny code reuse attacks (Tiny-CRA) that demonstrates the ineffectiveness of the threshold based detection methods. We show that with bare minimum assumptions, Tiny-CRA can reduce the size of a gadget chain in shuch a way that no distinction can be detected between normal behaviour of a program and a code-reuse execution. To do so, we exhibit our Tiny-CRA primitives and introduce a useful gadget set available in "libc. We demonstrate the effectiveness of our approach by implementing nine different shell-codes and exploiting real-world buffer overflow vulnerability in HT Editor 2.0.20.

1 Introduction

New protection mechanisms such as No-eXecute (NX) [1], Address Space Layout Randomization (ASLR) [2] and Stack Smashing Protection (SSP) [3] are among general defenses implemented in the recent operating systems. Since the wide adoption of NX-Bit defenses, attackers use exploitations in the form of code reuse. A primary form of Code Reuse Attacks (CRA) is Return-into-Libc [4], involves setting up arguments in the stack and redirecting the control flow to any desired function (e.g. execve()).

To generalize CRAs to execute any operation in a target system, Return Oriented Programming (ROP) is introduced [5]. The main idea of this method is to chain code snippets terminated in ret, called gadgets to control a victims machine. Several defenses like ROPdefender [6] and ROPGaurd [7] tried to detect ROP attacks by implementing a shadow stack and monitoring the call-ret pairs. Therefore, attackers started to use code snippets terminated in jmp to construct the gadget chain of an attack. This new technique is called Jump Oriented Programming (JOP)

* Corresponding author.
Email addresses: aliakbar.sadeghi@aut.ac.ir (A. Sadeghi), fr.aminmansour@aut.ac.ir (F. Aminmansour), shahriari@aut.ac.ir (H.R. Shahriari)
ISSN: 2008-2045

[8, 9].

To prevent CRA exploits, diversification approaches like ASLR dominate on a wide range of operating systems. ASLR randomizes the memory layout of a program by loading its binary and dynamic libraries in different base addresses each time [2, 10, 11]. However, attackers are still able to bypass ASLR by exploiting memory information leakage vulnerabilities [12].

A common feature among all types of CRAs is about the definition and usage of gadgets. Many detection mechanisms such as DROP, SCRAP, JOP-alarm as well as the state-of-the-art kBouncer family (e.g., kBouncer and ROPecker), consider this feature to distinguish a normal control flow from a code reuse one.

As Gktas *et al.* in [13] described, the values of two famous detection parameters, i.e., gadget length and number of gadgets used in an attack, are very important. Because conservative threshold values may produce several false positive alarms. On the other hand, optimistically determined thresholds may promote the false negative rate.

1.1 Contributions

In this paper [1], we intend to draw the layouts of Tiny-CRA that enables code reuse exploitations under thresholds. It shows that none of the threshold based detection methods are sufficient to stop exploitation in a real software. We intend to introduce different gadget structures, each of which cause to resemble an attack flow more similar to a regular program execution. There are also nine real-world exploits into a buffer overflow vulnerability in HT Editor 2.0.20 [15]. Finally, a comparison between available shellcodes and their transformation into our tiny mode is exhibited.

1.2 Outline

The rest of this paper is structured as follows: Section 2 provides a general background on code reuse attacks and existing countermeasures against them. Related work is in Section 3. Section 4 describes the high level idea of tiny code reuse attacks. Section 5 exhibits our exploitation technique in detail. In this section, we place our proposed gadget set for Tiny-CRA as well as three proof of concepts that bypass current threshold based detection methods. Section 6 exposes a comparison among different available CRA exploits and our tiny ones. Finally, Section 7 is devoted to drawing conclusions.

[1] This paper is an extended version of the earlier paper that has been presented in the 12th International ISC Conference on Information Security and Cryptology (ISCISC), Guilan, Iran, 2015 [14].

2 Background

In this section, we describe a general background on code reuse attack techniques and different countermeasures available on x86 architecture to prevent them.

2.1 Code Reuse Attacks

Memory corruption vulnerabilities like stack overflow [16], heap overflow [17], integer overflow [18], dangling pointer reference [19] and format string [11, 13] are among CWE/SANS top 25 Most Dangerous Software Errors in 2011 [20]. Malicious attackers are able to hijack the control flow of a program using these vulnerabilities. They inject bytes of any desirable code into memory and divert the control of a program so that injected bytes of shell-code can be executed. This is called Code Injection Attack, which is a popular way of executing any code in a target system [21]. To prevent these kinds of attacks, a new technology is used in CPUs called NX-Bit, which stands for No-eXecute. This technology segregates memory areas of processor instructions and data. Operating Systems with support for NX-Bit may mark programs data storage area as non-executable that then leads the processor to refuse executing any residing code in marked memory ranges [1]. The NX-Bit implementations in MS Windows and OpenBSD are called Data Execution Prevention (DEP) [22] and $W \oplus X$, respectively. $W \oplus X$ means a frame cannot be writeable and executable at the same time.

In the arms race between attacks and defenses, a new class of attacks called Code Reuse Attack (CRA) has been introduced. The main idea of these attacks is based on using the code snippets available in a target system instead of injecting any bytes of executable code.

We will continue this section with an overview on different types of available CRAs consist of Return-into-libc, return oriented programming and two models of jump oriented programming.

2.1.1 Return-into-Libc

Ret2Libc attack was introduced as the first instance of CRAs. Attackers exploit a stack overflow vulnerability of a program to change its control flow and execute any sequence of intended functions available in vulnerable programs libraries [4]. Figure 1 shows an example of Ret2Libc attack on Linux.

2.1.2 Return Oriented Programming

Ret2Libc attacks were limited to the logic and functionality of predefined functions available in the libraries. Therefore, attackers introduced a new type of

A)	Buff	Old %EBP	Old %EIP	Fnc pram#1	Fnc pram#2
B)	Dummy	Dummy	Fnc Address	Ret Address	Argument 1
C)	AAAAAAAA	AAAAAAAA	System()	Exit()	& /bin/sh

Figure 1. A) Stack layout before being rewritten, B) Ret2Libc structure in memory, C) Ret2Libc attack instance, which spawns a shell in Linux

CRAs called Return Oriented Programming (ROP), which is depicted in Figure 2. The main idea of ROP is based on chaining different sets of instruction sequences ending in ret to implement the whole attack behaviour. The code snippet sequences ending in branch instructions are called gadgets. Shacham *et al.* in [5] show that the `ret` is frequently applied in libraries, which makes attackers enable to find a lot of potential useful ROP gadgets. They provide a Turing complete gadget set for ROP as well.

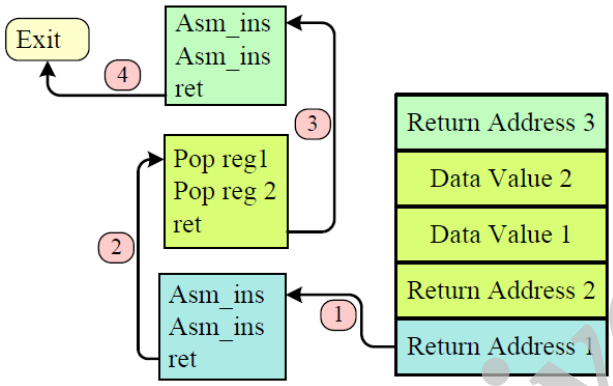

Figure 2. Return Oriented Programming

2.1.3 Return Oriented Programming without returns (a class of Jump Oriented Programming)

To escape from some of the ROP detection techniques, attackers tried to imitate `ret` by using another instruction set. Any `ret` retrieves the four-byte value at the top of the stack and sets the `eip` to that value. It also has to increase the value of stack pointer `(esp)` and makes it point to the DWORD above the current one. An indirect jump, then, is needed to change the control flow of a program to the next desired instruction. Satisfying these purposes is possible by using the `pop reg; jmp reg`; sequence as a trampoline [8]. Figure 3 shows the main structure of return oriented programming without returns. In step ❶, adversary hijacks the control flow of a program and redirects the instruction pointer to the first functional gadget (step ❷). All of the functional gadgets should branch to the trampoline, which is responsible to chain them sequentially (steps ❸ and ❺). In both ❹ and ❻, trampoline pops the next gadget address to the register `reg n` and then jumps to it. Finally, the last gadget branches to an exit [2], which terminates the program normally.

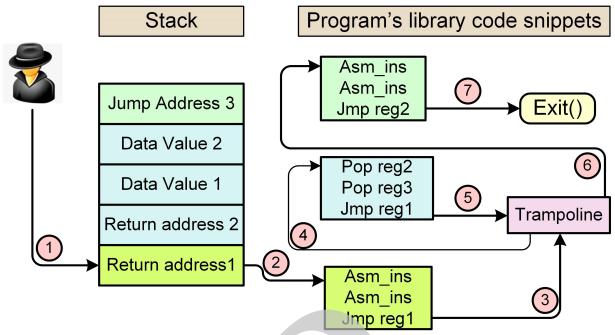

Figure 3. Return Oriented Programming without returns

2.1.4 Jump Oriented Programming

More recent attacks use a set of instructions that are terminated in a (doubly) indirect jump. This kind of attack is called Jump Oriented Programming (JOP). Figure 4 shows the JOP technique introduced by Bletsch *et al.* in [9]. First of all, an initializer gadget is needed to load registers with suitable values. The final instruction of initializer is a doubly indirect jump, which diverts the control to the first functional gadget (step ❷). It should be noted that the final instruction of all functional gadgets should branch back to the dispatcher (steps ❸, ❻ and ❾). Dispatcher is responsible to update the pointer register and jump to the next functional gadget by a structure like `add reg, 4; jmp DWORD [reg]`; (steps ❹ and ❼). In step ❺, the gadget loads two registers with appropriate data values available among jump addresses in memory. Finally, the attack terminates normally in 10 by executing an exit.

2.2 Countermeasures

This section is a brief review on different detection methods of Code Reuse Attacks available on x86 systems.

2.2.1 ASCII Armor

ASCII Armoring is a technique introduced to obstruct Ret2Libc attacks. It changes the addresses of system libraries in memory so that they contain at least a

[2] It should be mentioned that the Exit in Figure 2, 3 and 4 can be a real exit function, the next instruction address in a regular program control flow or an interrupt that calls a system call.

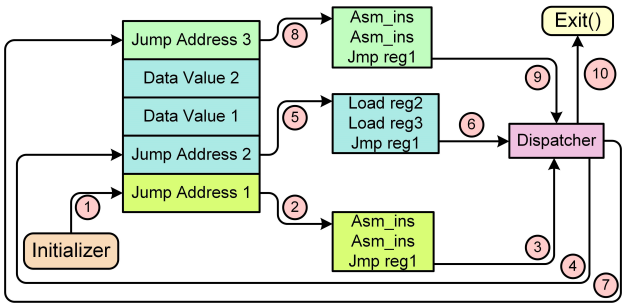

Figure 4. Jump Oriented Programming using Dispatcher gadget

NULL byte. Therefore, string manipulation functions such as `strcpy()` cannot emplace function addresses in memory. This mechanism has been bypassed by Return to PLT attack, which uses Procedure Linkage Table functions loaded in the binary instead of libc [23]. Note that in the rest of this paper, CRA and ROP/JOP are used interchangeably.

2.2.2 DROP

Chen *et al.* in [24] presented a tool DROP that instruments the binary dynamically to detect ROP malicious code. ROP has an intrinsic difference from a normal program: (1) uses short code snippets ending in `ret`, which is called gadget; (2) executes a sequence of gadgets in specific memory space like libc. Therefore, two thresholds should be considered: T0 as the maximum size of gadget length and T1 for the minimum number of gadgets used in a gadget chain. While DROP is a threshold based detection technique, the false positive and false negative rates depend on the value of T0 and T1. It has been shown that by determining the T0 equal to 5 and T1 equal to 3, no false positive and false negative would be raised. The weaknesses of this method include its 5.3X overhead and inability to detect JOP attacks. Also, it can be bypassed by using delay gadgets in the attack chain.

2.2.3 ROPscan

Polychronakis *et al.* in [25] presented a tool ROPscan to detect ROP attacks dynamically. It uses a 4 byte sliding window, which advances byte by byte on input buffer to find a valid address in programs memory address space. When a valid address is found, `eip` and `esp` would be set to the relative values and an emulator starts the execution. The execution continues normally unless one of the following conditions is reached: (1) instructions of a gadget transfer the control to an invalid address; (2) an invalid or privileged instruction encountered; (3) current gadget length crosses the threshold; (4) total number of executed instructions reaches an overall threshold.

2.2.4 ROPdefender

In 2011, Davi *et al.* presented a tool called ROPdefender [6]. The main idea of ROPdefender is based on a fact that each `ret` instruction should be paired with a `call`. Thus, by performing each `call`, a copy of the return address is kept in a dedicated memory area, referred to as shadow stack. Upon a return, it checks whether the top of the stack and shadow stack are the same. Hence, if any return address on the stack has been modified, the attack would be detected. Figure 5 depicted the ROPdefender detection mechanism.

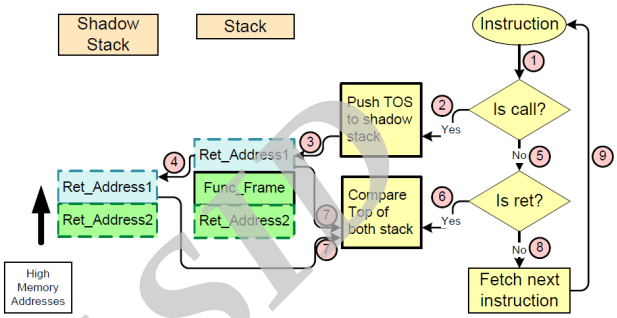

Figure 5. ROPdefender detection mechanism

ROPdefender tool is designed and implemented on top of the Pin framework [26], which provides a just in time binary instrumentation. The weakness of ROPdefender is that it has a high performance overhead, which is inevitable in using instrumentation tools.

2.2.5 JOP-alarm

JOP-alarm is an anomaly based detection method proposed in 2013 by Yao *et al.* to detect Jump Oriented Programming attacks. The algorithm is based on scoring the instructions executed as follows: (1) the score is incremented by `step_up_values` based on jump target distance, e.g., to reduce the false positive rate, when any indirect jump/call is encountered; and (2) the score is decremented by `step_down_values` for all other instructions. Thus, the algorithm is completely tune-able to turn more or less aggressive [27].

$$score_{inc} = f_{\texttt{ind\_jmp/ind\_call}}(\texttt{jumptarget\_dist}, \texttt{step\_up\_values}) \quad (1)$$

$$score_{dec} = f_{\texttt{otherinst}}(\texttt{step\_down\_values}) \quad (2)$$

If the total score becomes greater than a value `jop_threshold`, a detection alarm will be raised. Through careful analysis of JOP attacks in [9, 28], the `step_up_values` determined to be 20 and the `step_down_values` to be 1. While they assume that at least six gadgets are needed to set up an attack, `jop_threshold` is considered to be at least 120.

2.2.6 SCRAP

SCRAP is a signature based protection from code reuse attacks that can be implemented entirely in hardware. The first idea was to use two thresholds: `T1` as gadget length and S as the number of consecutive gadgets. Thus, if a delay gadget which is longer than `T1` was executed among a gadget sequence, the counter of gadgets would be reset. To prevent attackers from this circumvention, Kayaalp *et al.* in [28] considered another threshold `T2` for the maximum length of delay gadgets. They also prudentially determined that the `T1` to be 7, `T2` to be 20 and an `S` of 4. Therefore, if at least 4 gadgets with length of at most 7 is executed, the attack will be detected. Furthermore, executing a delay gadget up to the size of 20 will not reset the gadget counter. Based on our careful analysis on this system, adjusting S to 4 exacerbate the false positive issue. A realistic careful value for S is to be 7.

2.2.7 ROPDetector

ROPDetector is a dynamic detection method against ROP and JOP, which is implemented on Pin framework. ROP instruction sequences possess some characteristics that differentiate them from a normal control flow of a program. First, they use short length instruction sequences while the side effects of long ones are high. Second, gadgets are selected from both intended and unintended instructions. Third, as the function prologue manipulates top of the stack, ROP will avoid it; so, no ROP gadget address can be a function entry address [29]. For JOP, Huang *et al.* believe that legitimate `jmp` instructions in a normal program control flow are taken only within a function boundary. This method has many ways of circumventing, high performance overhead, e.g., 3.5x, and high false positive rate.

In the rest of this part, we will illustrate two defense mechanisms pioneered by kBouncer that nowadays are known as the most practical countermeasures against Turing complete CRAs. A key characteristic of ROP/ JOP attacks is that they all use a number of chained gadgets each of which indicates a part of attack. Like previous methods, two threshold values are considered: T_{GL} as the maximum size of Gadget Length and T_{CG} as the minimum number of Chained Gadgets. Furthermore, kBouncer-family methods use a new set of registers available in modern Intel CPUs called Last Branch Record (LBR), so that, operating system can log the target of last n indirect branches taken by a program.

2.2.8 KBouncer

KBouncer is triggered every time a sensitive API call like `VirtualProtect()` , `CreateProcess()`, etc. is executed. The detection algorithm scans LBR registers to find out if any sequence of CRA gadgets has made the API call or not. If the result is true, it terminates the running process. The main idea of kBouncer is a combination of ROPdefender and DROP with lower overhead through its implementation. In a typical program control flow, the targets of `ret` instructions are located right after their respective calls. In contrast, ROP attacks transfer control flow from the end of one gadget to the beginning of another which is unlikely to be preceded by a `call`. Therefore, with a right configuration in kernel level, it is easy to distinguish `ret` instructions of ROP and legitimate rets of a benign program at runtime. This constraint in kBouncer is more relaxed as there is no need to check the target address of `ret`. In other words, ensuring that each `ret` is following a `call` is enough. Gadgets begin after intended or unintended `call` instructions are call-preceded gadgets. While attackers were able to use call-preceded gadgets to construct a ROP payload, a more conservative approach has been followed. In addition, it was still possible to use Jump Oriented code to increase complexity of attack and evade the mentioned detection method [30].

To account these sorts of exploits, a second mechanism is introduced, which is based on T_{GL} and T_{CG}. Every uninterrupted instruction sequences less than T_{GL} that ends in a branch is assumed as a potential gadget through an offline analysis. In this case, T_{GL} = 20 and T_{CG} = 8. Hence, If the ROP/JOP chain of gadgets exceeds the threshold of 8, the attack will be detected [30].

2.2.9 ROPecker

ROPecker is a variation of kBouncer family, which is provoked more often through a new triggering mechanism. It uses a sliding window maintained the executable code (between 2 and 4 pages), while all other code pages are non-executable. If an attacker diverts the control flow to out-side of the sliding window, a check is set off. The detection algorithm uses past and future scan results to detect if there is at least 11 ROP/JOP gadgets, e.g T_{CG} = 11 to be a safe choice. In particular, a gadget is a sequence of at most 6 instructions, e.g. T_{GL} = 6, ending in branches except direct ones. Therefore, using any direct branches among other branches would reset the gadget counter value [31]. The past scan is possible by tracking the branch and branch target addresses logged in LBR registers. Then, a future scan is occurred by inspecting the addresses stored in stack. This inspection is

done through emulating potential gadgets extracted through an offline static analysis of applications. If the total number of past and future gadgets exceeds T_{CG}, it would raise a detection alarm.

3 Related Work

CRAs such as Ret2LibC [4], ROP [5] and JOP [8, 9] are the most popular techniques available in x86 architectures. According to the common behaviour of ROP and JOP, two thresholds are defined: one for gadget length and the other for size of gadget chain. Most of detection methods consider these thresholds among other properties. Some of them like SCRAP [28], JOP-alarm [27], kBouncer [30] and ROPecker [31] try to find the best values of these thresholds to minimize both false positive and false negative rates. Through careful analysis on different benchmarks, they reveals that adjusting the threshold of gadget chain length less than 6 is hardly realistic. Pappas *et al.* in [30] choose the values of $T_{GL} = 20$ and $T_{CG} = 8$ and Cheng *et al.* [31] choose the values of $T_{CG} = 11$ and $T_{GL} = 6$. Moreover, Kaayalp *et al.* in [28] claim that even the most basic attack requires at least 6 gadgets. Our endeavour illustrates that it is possible to launch a lot of efficient attacks under these safe choices, which makes it impossible to differentiate a normal behaviour from malicious one by means of thresholds. Note that there are also several related attempts done by Sadeghi *et al.* in previous works including [32] and [14]. Moreover, other work done by Aminmansour *et al.* in [33] and [44] are done on mobile devices with ARM architecture. Not surprisingly, there are also other proposals to bypass current detection methods. The recent attempts concentrate on bypassing kBouncer family as the most practical defense mechanism available. For this purpose, attackers usually use the following tricks: 1. perform a direct branch after execution of each 5 gadgets to bypass ROPecker. 2. Use a delay gadget that can reset the LBR stack and flush it. 3. Apply call-preceded gadgets, e.g ROP gadgets starting with a call [34]. Obviously, our approach to bypass current detection methods is completely different from them.

4 Tiny Code Reuse Attacks

Regarding Section 2.2, most of the Code Reuse Attack defenses are based on determining some threshold values. The purpose of Tiny Code Reuse Attacks is to decrease the number of gadgets enrolled in an attack scenario as much as possible to bypass available detection methods. In this section, we describe the structure of Tiny-CRA, techniques, circumstances and useful gadgets available on Linux platforms.

4.1 The High-level Idea

A common behaviour among all of ROP and JOP attacks is the occurrence of functional gadgets including register loaders and operational ones. Register loaders are responsible to update the content of general purpose registers occasionally with appropriate values in the course of main attack. In addition, both types of JOP attacks need to execute a linker gadget, e.g., trampoline and dispatcher, between each of two functional ones. Furthermore, attackers need to use a number of NULL-Writer gadgets to zero out suitable bytes of memory space of attack. Because most of the vulnerable C functions like `strcpy` and `strcat` are not allowed to copy zero bytes of shell-codes. If this were to happen, it would end up inferring an incorrectly parsed string from the victims machine.

Considering all of the above, size of gadget chain in any preliminary attack might go beyond the current determined thresholds of detection methods. Tiny-CRA decreases the number of gadgets in a chain to implement an attack under thresholds. Short size of gadget chain makes it difficult to distinguish between a regular program execution and an attack.

The principles of a Tiny-CRA is depicted in Figure 6. It shows an abstract view of a programs memory space. First of all, an adversary hijacks the control flow of a program through a memory corruption error and diverts it to the initializer gadget. The Initializer gadget loads all of the general purpose registers with appropriate values. After that, a number of operational gadgets are executed consecutively to advance the stage of attack. Finally, the last gadget will load an appropriate system call number into `eax` and commit the kernel to execute it. Note that due to the

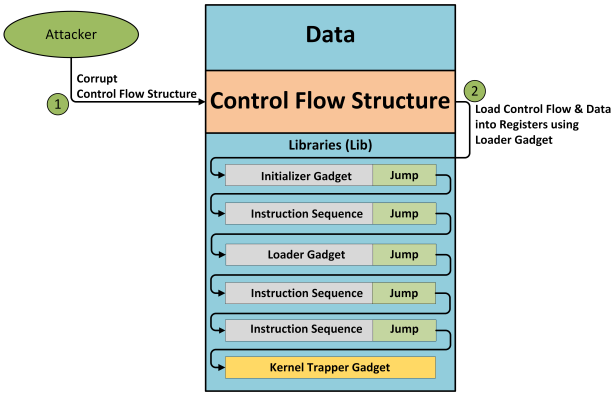

Figure 6. Big picture of Tiny Code Reuse Attacks

necessity of loading new values into registers, another loader gadget can be executed among operational ones. There are just 8 general purpose registers in 32-bit Intel x86 architecture consist of `eax, ebx, ecx, edx,`

esi, edi, ebp, esp. esp is the stack pointer that always points to the top of the stack. Hence, execution of a limited number of consecutive operational gadgets is possible by using the remained 7 general purpose registers. Tiny-CRA method dwindles the need of any trampoline or dispatcher in JOP attack structures. It means that it is not necessary to execute a trampoline or dispatcher between every two functional gadgets. Likewise, functional gadgets are not confined to end in an (doubly) indirect jump to a special register. There are two main reasons for execution of dispatcher and trampoline gadgets. Firstly, the proposed JOP models have been supposed to imitate the convention of ROP attacks, which are based on the specific behaviours of ret instruction. Secondly, distinguishing between the two roles of advancing and executing the steps of attack, constructs a clear structure to simplify the development of automatic methods to generate a JOP shell-code. However, it is obvious that there is no need to execute any linker between two functional gadgets. Different types of functional gadgets ending in ret, jmp or call instructions can be stitched directly together in an appropriate order without the intermediation of any linker. Eliminating trampoline and dispatcher gadgets in an attack convention reduces the number of gadgets nearly to the half of the common JOP attacks. Furthermore, it broadens the pool of potential JOP gadgets since no binding to a specific register must be considered in the last instructions of JOP functional gadgets. In Tiny-CRA model, attacker can utilize a combination of any functional gadgets ending in ret, jmp and call. Therefore, usually the length of a functional gadget and consequently the number of operations in it, increases during the execution of a single gadget. Moreover, a combinatory attack model meets and satisfies the prerequisites of multi system call attacks. Also, note that applying a combination of call-ret gadgets and JOP ones makes it possible to bypass preliminary control flow integrity (CFI) defenses such as ROPdefender. In the next section, we elaborate some details and examples about Tiny-CRAs.

5 Instantiation on INTEL X86

5.1 Assumptions and Adversary Model

To be as pragmatic as possible, we have made assumptions below to define the adversary model:

(1) The target platform may enforce the $W \oplus X$ security model to prevent code injection attacks.
(2) Adversary cannot copy NULL bytes through vulnerable functions.
(3) Adversary can only derive usable gadget set from a single shared library such as libc.
(4) Other protection mechanisms like ASLR, stack canaries, boundary checkers and pointer encryption are typically bypassed in the first stage of attack. In particular, it is possible to bypass ASLR by using offset2lib attack [35]. The goal of the attack is to obtain an address which belongs to the application code. Then by obtaining the memory mapped areas of all libraries, attackers are able to bypass ASLR in GNU/Linux systems. Also, it is good to note that while ASLR is vulnerable to memory disclosure attacks, an attacker can leverage some ROP gadgets to find out the addresses of memory space [10, 36, 37]. An effective implementation of this technique might be applicable in attacks with lower gadget number to prevent crossing available thresholds.

5.2 Gadget Set

To reduce the size of gadget chains in an attack, we will introduce three types of gadgets here.

5.2.1 Initializer/Loader Gadget

Initializer or Loader is responsible to load all of the general purpose registers using just one gadget. This can be achieved through POPA or POPAD instructions to load all of the eight general purpose registers in order of edi, esi, ebp, esp, ebx, edx, ecx, eax . One may conjecture that using POPA(D) instruction would destroy the latest state of attack in registers which is constructed during execution of previous gadgets. It should be mentioned that an adversary can construct her exploit in memory in any desired order. The only problem of rewriting all registers is about NULL values that we cannot copy them directly through most of the vulnerable functions. Section 5.2.2 demonstrates that how we will deal with this issue. Through a precise scrutiny in Shell-Storm, a dataset of available shell-codes, we found out that the shell-codes with single system calls use eax, ebx, ecx and edx to set the system call number and pass the input arguments. So, a better choice of Initializer/Loader is a popa-ret sequence or a popa-jmp sequence, which is ending in an indirect jump to ebp, edi, esi or eax . Note that while the final gadget would load eax, this register could be involved in any operations during the execution of gadgets. Figure 7 shows the algorithm of finding Initializer/Loader gadgets. The first step is to define several sets including: 1) POPs, which consists of all variations of popa instruction family; 2) validreg, which consists of four admissible above mentioned registers as the target of indirect jumps; 3) and finally validins that consists of all of the permissible instructions with definitely no further impact on currently loaded general purpose registers. An obvious attribute to consider a gadget as a potential Initializer/Loader

Procedure HuntingTinyJOP*InitializerGadgets* (*G*)
1. *POPs* ← (popa, popad, popal)
2. *F* ← (First instruction of gadget)
3. *L* ← (Last instruction of gadget)
4. *Validreg*←(eax, esi, edi, ebp)
5. *Validins*←(test, stc, nop, cmc, cld, std, das, cmp, into, sahf, fucomp, fsubp, fdiv, arpl)
6. **If** (*F* ∈ *POPs*) **then**
7. **If** ((*L*.operand has one register)∧ (*L*.operand ∈ *Validreg*) ∧ (*L*.operand is a *jump*)) ∨ (*L*.operand is *ret*) **then**
8. **for all** instruction *I* ∈ *G*, such that I ≠ F and I ≠ L **do**
9. **if** (*I* ∉ Validins) **then**
10. **return false**
11. **end if**
12. **end for**
13. **return true**
14. **end if**
15. **end if**

Figure 7. The algorithm of finding Initializer/Loader gadgets.

is that the first instruction (e.g., F) should be one of the `popa` family and the last instruction (e.g L) should be an indirect jump or a return. Therefore, the following conditions must be met: imprimis, we check the first instruction of a gadget whether it is a member of POPs set or not. Then, we check for the last instruction to use one of the valid registers for Initializer/Loader, which is mentioned above. Also, we need to check for inner instructions not to be one of the irreversible instructions. By irreversible instruction we mean those instructions that might manipulate the content of our loaded general purpose registers. Here is a sample result of running the algorithm on libc-2.19.so:

```
0x00087a93: popad; jmp DWORD [esi+0x0F];
```

5.2.2 NULL-Writer Gadgets

Common `NULL` writer gadgets usually increase the size of gadget chains considerably. Table 1 shows a sequence of 13 gadgets that intends to write only three `0x00` bytes in the address of `ebx-0x17bc0000`. g00 loads `eax` with `0xffffffff`, so that increasing a unit to `eax` in g01 will convert it to `0x00000000`. Therefore, we intend to use other alternatives to make `NULL` bytes.

NULL-Maker Gadget applies arithmetic operations in order to produce `NULL` values in memory or registers. By determining appropriate values for source and destination operands, desired number of `NULL` bytes can be made by only one gadget. Two examples of these gadgets are shown below:

```
0x001a3ce1: add eax, edi; jmp DWORD [esi];
0x001999ed: add DWORD [esi], edi; cmc; jmp DWORD [edx];
```

Toggling Gadget is another alternative for NULL-Writer gadgets. These gadgets use `XOR` operation to construct any desired value consists of `NULL` bytes in registers or memory addresses. For instance, here is a toggling gadget:

```
0x0019d92d: xor eax, edi; cmc; jmp DWORD [edx];
```

Masking Gadget utilizes `AND` instruction to reset desired bytes of registers or memory. The difference between toggling and masking gadget is that the former is able to set and reset each bit, while the latter one is just able to reset bits. An example of masking gadgets is as follows:

```
0x0019bd61: and DWORD [eax-0x0B], ebp; jmp DWORD [edx];
```

Table 1. A regular NULL-Writer gadget chain that puts three bytes of 0x00 in memory [9].

g00	`popa; fdivr st(1), st; jmp DWORD [edx];`
Dispatcher	`add ebp,edi; jmp DWORD [ebp-0x39];`
g01	`inc eax; fdivr st(1), st; jmp DWORD [edx];`
Dispatcher	`add ebp,edi; jmp DWORD [ebp-0x39];`
g02	`mov [ebx-0x17bc0000], ah; stc; jmp DWORD [edx];`
Dispatcher	`add ebp,edi; jmp DWORD [ebp-0x39];`
g03	`inc ebx; fdivr st(1), st; jmp DWORD [edx];`
Dispatcher	`add ebp,edi; jmp DWORD [ebp-0x39];`
g04	`mov [ebx-0x17bc0000], ah; stc; jmp DWORD [edx];`
Dispatcher	`add ebp,edi; jmp DWORD [ebp-0x39];`
g05	`inc ebx; fdivr st(1), st; jmp DWORD [edx];`
Dispatcher	`add ebp,edi; jmp DWORD [ebp-0x39];`
g06	`mov [ebx-0x17bc0000], ah; stc; jmp DWORD [edx];`

5.2.3 Kernel-Trapper Gadget

A system call is a request for a service from a program to the kernel. A service is generally a privileged task that only the kernel can do. However, there are times that one needs to make system calls explicitly. Therefore, libc provides some syscall functions. Trap Gates and Interrupt Gates are entries of the Interrupt Descriptor Table (IDT). They contain Segment Selector (SS) address and an offset inside this segment that points to interruption or exception handler. Interrupt gates clear the Interrupt Flag (IF) bit when an interrupt occurs, disabling further hardware interrupts. Trap gates leave the IF bit unchanged [38]. There are three ways of implementing a System Gate on x86 architecture consist of `int 0x80`,`syscall` and `sysenter` . A System Gate is a Trap Gate accessible by user mode programs. Additionally, there is `call DWORD PTR gs:0x10`, which is a call to the vdso area in 32bit Linux. The superficial similarity between all of them is that they need to set up`eax` with an appropriate system call number before [38]. `int 0x80` is

particular among other Trap Gates because of calling interrupt service routine. It also sets up saving register values on the stack as interrupt stack frame. These actions are time consuming. Therefore, Sysenter and `call DWORD PTR gs:0x10` are introduced to make a system call faster. `sysenter` applies Vsyscall technology, which is an interface for 64 bit Linux. It was created to speed up certain time-sensitive system calls, e.g. `time()`, `gettimeofday()` and `getcpu()`, so that, an application can simply jump to static addresses in a static page set up by the kernel. While the only compiled version of the libc has to run equally well on all CPU versions (486, 586 or 686), there was a need for an abstraction layer called by the libc, which would choose the best mechanism at runtime. The Virtual Dynamic Shared Object (VDSO) is a small shared library that the kernel automatically maps into the address space of all user-space applications. This way, you can code in the normal way using standard functions, and C library will take care of using any functionality that is available via VDSO. Note that the VDSO area has moved, while the vsyscall page remains at the same location. The location of the vsyscall page is nailed down in the kernel Application Binary Interface (ABI), but the VDSO area, such as most of other areas in the user-space memory layout, has its location randomized every time it is mapped. We scrutinized the Shell-Storm Linux shellcode repository [39] to find out the distribution of its shell-codes based on the number of system calls involved. The result is depicted in Figure 8. Out of 209 plain Linux shell-codes available in Shell-Storm, 81 are mono system calls, 52 are dual system calls and the remainders are multi system calls.

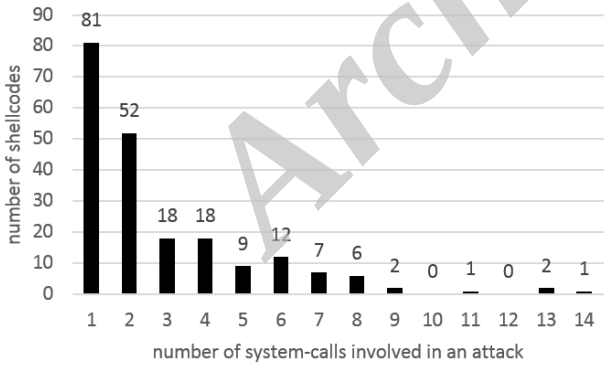

Figure 8. Distribution of shell-codes based on the number of system calls applied in each attack of shell-storm repository.

The gadget set of mono system call attacks is shown in Table 2. We call them Kernel-Trapper Gadgets that are only available in libc. They use vdso trampoline, e.g., `call DWORD PTR gs:0x10`, following an instruction that sets an appropriate value in eax. Thus, setting up eax value and performing a system call become feasible through executing only one gadget.

Table 2. Kernel-Trapper Gadgets in libc

b673b: mov eax, 0xb ;	da6c6: mov eax, 0x1 ;
call DWORD PTR gs:0x10	call DWORD PTR gs:0x10
db0ba: mov eax, 0xf ;	dc0ba: mov eax ,0x3f ;
call DWORD PTR gs:0x10	call DWORD PTR gs:0x10
db707: mov eax, 0x3 ;	dd696: mov eax , 0xa ;
call DWORD PTR gs:0x10	call DWORD PTR gs:0x10
ec546: mov eax, 0x6e ;	e480a: mov eax, 0x4a ;
call DWORD PTR gs:0x10	call DWORD PTR gs:0x10
dd726: mov eax, 0x28 ;	e48ba: mov eax, 0x79 ;
call DWORD PTR gs:0x10	call DWORD PTR gs:0x10
e4cc2: mov eax, 0x58 ;	ec83a: mov eax, 0x34 ;
call DWORD PTR gs:0x10	call DWORD PTR gs:0x10
db7af : mov eax, 0x4 ;	e4ba0: mov eax, 0x24 ;
call DWORD PTR gs:0x10	call DWORD PTR gs:0x10
12c0a2: mov eax, 0x25 ;	
call DWORD PTR gs:0x10	

5.3 Bypassing Current Detection Methods

In this section, we establish three Proof of Concept attacks constructed on Linux 14.04 LTS to bypass current threshold based detection methods. These real world attacks exploit a buffer overflow vulnerability in HT Editor 2.0.20 (CVE-2012-5867) [40]. We exploit this vulnerability using Tiny-JOP to run our desired operation on the target system. We have used code snippets available in libc-2.19.so[3] to construct exploits of spawning a shell, setting system time to 0 and exit as well as a reverse shell. It is good to note that we sketch the stack of each attack after being rewritten, stack addresses and sequence of gadgets to deliver an explicit comprehension of our design.

5.3.1 Spawning a Linux shell

One of the typical attacks on Linux operating systems is to spawn a shell in a local machine by executing the execve system call. `execve()` executes the program pointed to by filename. Filename could be an executable binary, e.g. `/bin/sh` . `execve` has the system call number of `0x0b` and its definition is as follows:

```
int execve(const char *filename, char *const
argv [], char *const envp[]);
```

[3] The MD5 of this Libc version is: e35a3b9d92f436e7765869bc498567d6.

argv and envp are two other input arguments that pass argument strings as well as environment key=value strings to the function. Both of them must be terminated by a NULL pointer. In our attack scenario, we are not going to pass any argument variable or environment string to execve() [41]. Putting them all together leads us to the following state (shown in Table 3) for eax, ebx , ecx and edx registers before performing any syscall (in our case by VDSO trampoline).

Table 3. Appropriate state of registers for spawning a shell.

Registers	eax	&ebx	&ecx	&edx
Values	0x0b	/bin/sh	NULL	NULL

As shown in Figure 9, attacker is able to spawn a shell by applying only two gadgets. Initializer is responsible to load ebx,ecx and edx with the addresses of unintended bytes of string "/bin/sh", and two available NULL DWORDs, respectively. Then, the Kernel-Trapper will load eax with 0x0b and make a system call.

Obviously, using only two gadgets in such an attack will bypass all of the threshold based detection methods, specifically the state-of-the-art kBouncer and ROPecker. Moreover, due to the use of JOP gadgets in an attack, ROP detections based on implementing a shadow stack, e.g., ROPdefender, are bypassed.

5.3.2 Setting System Time to 0 and Exit

Here, we are going to demonstrate an attack-like with more than a system call using our Tiny-CRA method. Attacker intends to set the time of a system to zero and exit normally. To zero out the time of a system, one can invoke stime system call through sys_stime kernel routine. The syntax of this routine is as follows: sys_stime(int *timeptr)

The only input argument of this routine is an integer pointer to a desired time. The suitable state of attack for the first system call is to set ebx to the address of time and eax to 0x019. The two first gadgets depicted in Figure 10 explain how we construct such an attack. ebx points to a NULL value in libc. When the control comes back from the kernel mode, it will point to the instruction after vdso trampoline and will continue the execution of current routine. Implemented system call routines in libc contain different direct and indirect branches. Eventually, the execution of final return from the routine will give the control back to the attacker. She can put her desired return address in a right place on the stack and start the second stage of her attack. Therefore, fourth gadget is related to performing an exit(-1) as the second stage of this attack. The ebx register points to 0xffffffff and finally the Kernel-Trapper performs a system call. As we have mentioned in previous section, the number of gadgets used to implement current attack is less than available thresholds. Therefore, most of the threshold based detection methods would be bypassed. While we need to return from a system call routine to redirect the control to the next gadget, an unpaired ret with no preceded call would occur. Hence, return address checkers such as ROPdefender could still be able to detect the attack. The more relaxed return address checkers that only check whether the ret and call instructions are paired, could be bypassed by using a call preceded Initializer/Loader or any previous operational gadgets executed before the current one which calls vdso to execute the first system call. Also, it is possible to use a sequence of call-instructions-popa-nondestructive instructions-jmp gadget before executing the first operational system call one.

5.3.3 Reverse Shell Attack

A reverse shell is a reverse connection created from a target machine to communicate back to the attackers system. In other words, the attacking machine has to listen constantly on a specific port to catch up any connection received. After the connection is created, you can launch commands back from the connection destination machine (e.g., attackers system) to the connection originating machine (e.g., target system) with the credentials of the connection creator (e.g., target system). This is achieved by using a remote shell program or command execution. This part illustrates how to spawn a shell on a remote machine through a reverse bind for our local shell. For the sake of simplicity, we assume that the target has the netcat installed. The -e flag in traditional netcat will execute anything and bind it to the connection. Attacker needs a shell on her local machine running netcat -lvp ProtNo to begin listening to inbound connections. Also, she needs to execute the ROP/JOP equivalent of the following C code in the target system:

```
char *command[] = "/bin/netcat", "-e",
"/bin/sh", "IPAddress", "PortNo";
```

In our case, PortNo is 6666 and IPAddress is 192.168.47.1. Table 4 shows the right state of registers before performing execve system call. Figure 11 shows the Tiny-JOP sequence of gadgets. In the first step, Initializer loads eight general purpose registers with values appropriate to the execution of next gadgets. Second gadget will make two NULL DWORDs in memory as the termination of the strings ‘‘6666" and ‘‘192.168.47.1". Likewise, third gadget will make a Null DWORD in the address of next popping edx. Then, the Loader sets suitable values in all of the eight registers again. Therefore, ebx will point to

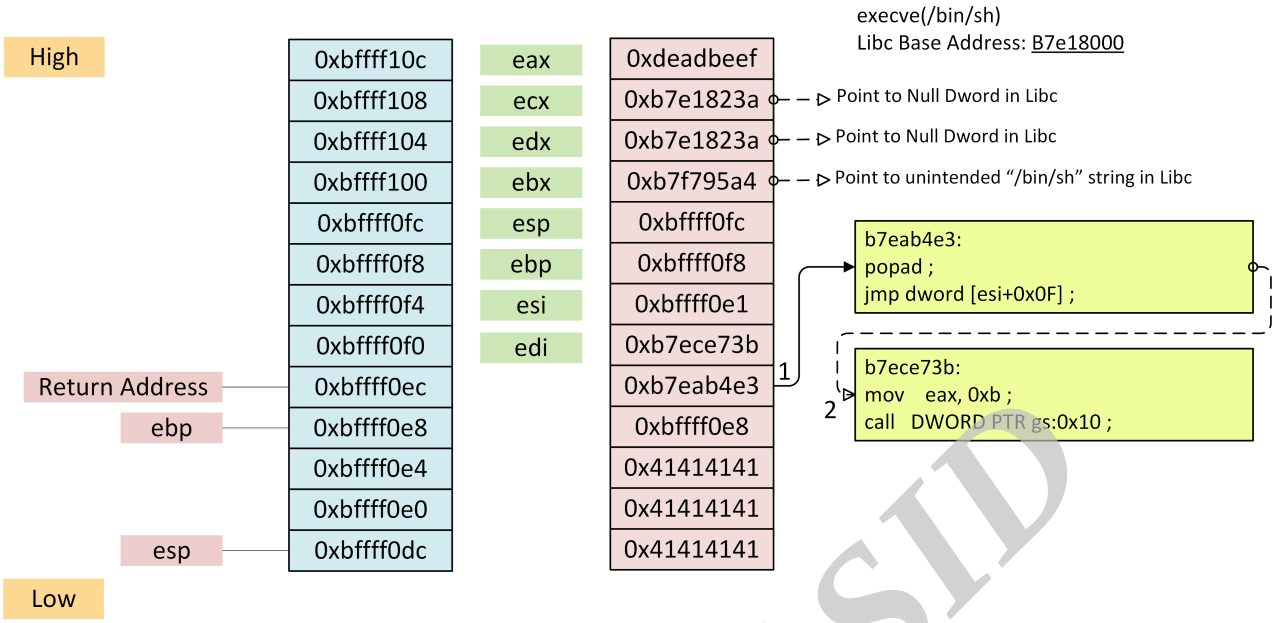

Figure 9. Spawn a shell.

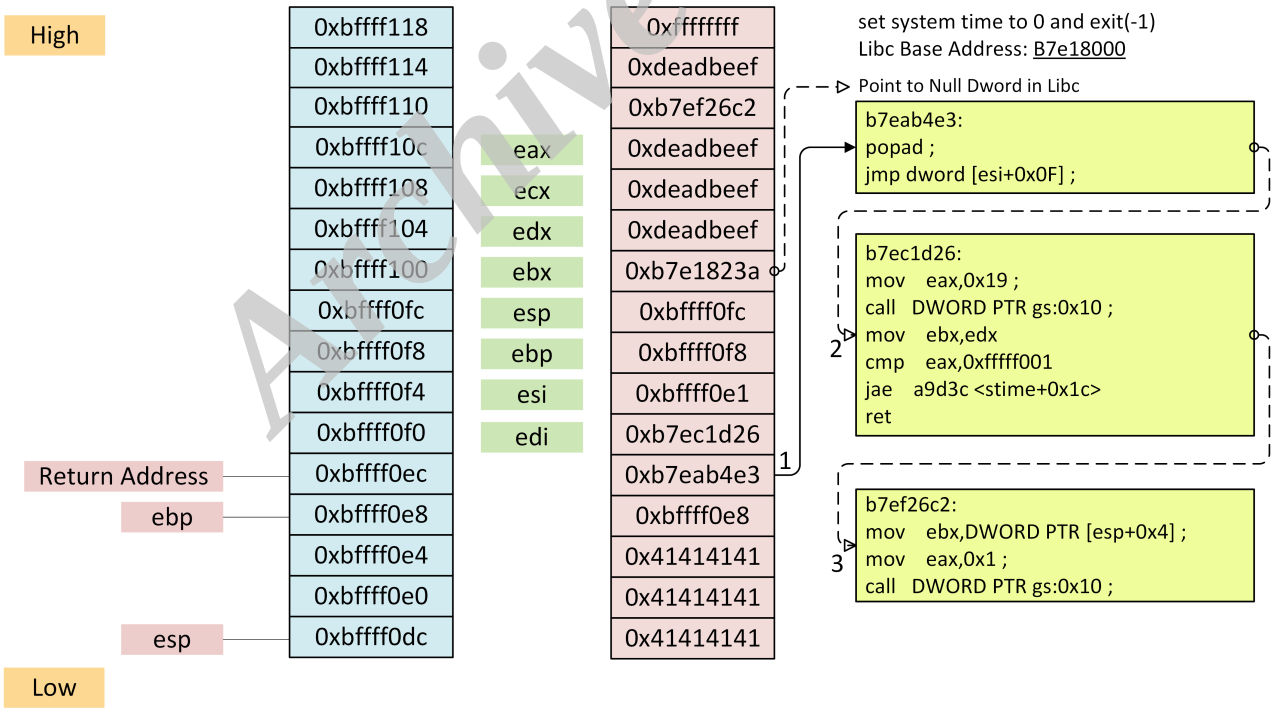

Figure 10. A dual system call attack that sets system time to 0 and exit.

the beginning of the input string, e.g., `command[0]`, and `ecx` will point to the array of input string addresses. Gadget 5 put the `NULL` DWORDs after strings of `"//bin/netcat"` and `"//-e//bin/sh"`. Finally, Kernel-Trapper sets the `eax` and performs a system call.

Table 4. State of registers for reverse shell attack.

Registers	`eax`	`&ebx`	`&ecx`	`&edx`
Values	`0x0b`	`"//bin/netcat"`	addresses of	`NULL`
		`"//-e//bin/sh"`	input strings	
		`"6666"`		
		`"192.168.47.1"`		

Performing this attack by using only 5 gadgets is under every thresholds determined and configured in aforementioned detection methods. Also, all of the gadgets here are terminated with an indirect jump. Therefore, call-ret checkers could not detect the attack. We claim that both of kBouncer and ROPecker are inefficient against this attack.

6 Evaluation

A comparison between published JOP shell-codes and our tiny method demonstrated in Table 5. It shows that transformation of shell-codes into tiny form will reduce the size of exploits to more than one a third. We compare the shell-codes by their size in bytes as well as the number of gadgets involved.

7 Conclusion

In this paper we explored the feasibility of bypassing all of the threshold based detection methods, including state-of-the-art ROP/JOP defenses like kBouncer and ROPecker, against code reuse attacks. Essentially, all of these defenses consider CRA common behaviour that depends on determination of gadget chain length as well as size of each gadget. A conservative policy may cause a lot of false positive alarms while optimistically determined thresholds ignore many attacks. We demonstrated how to reduce the number of attack gadgets in order to make the distinction between a program normal behaviour and an attack impossible by considering threshold based checks. For this purpose, we introduced a gadget set consists of Initializer, NULL writers and Kernel-Trapper among other tricky methods. Additionally, we eliminate the need for trampoline and dispatcher in JOP attacks. Putting these all together brings, approximately, a diminution more than a third of usual available gadget chains that makes it possible to bypass DROP [24], ROPscan [25], ROPdefender [6], JOP-alarm [27], SCRAP [28], ROPDetector [29], kBouncer [30] and ROPecker [31].

References

[1] E. Grevstad, "CPU-based security: The NX bit," Disponvel Line Em Julho De, 2004.

[2] P. Team, PaX address space layout randomization (ASLR). 2003.

[3] H. Etoh and K. Yoda, GCC extension for protecting applications from stack-smashing attacks. 2000.

[4] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in Recent Advances in Intrusion Detection, 2011, pp. 121141.

[5] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in Proceedings of the 14th ACM Conference on Computer and Communications Security, New York, NY, USA, 2007, pp. 552561.

[6] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 4051.

[7] I. Fratric, Runtime Prevention of Return-Oriented Programming Attacks. June, 2012.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented Programming Without Returns," in Proceedings of the 17th ACM Conference on Computer and Communications Security, New York, NY, USA, 2010, pp. 559572.

[9] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented Programming: A New Class of Code-reuse Attack," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, New York, NY, USA, 2011, pp. 3040.

[10] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in Security and Privacy (SP), 2013 IEEE Symposium on, 2013, pp. 574588.

[11] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in Proceedings of the Second European Workshop on System Security, 2009, pp. 18.

[12] F. J. Serna, CVE-2012-0769, the case of the perfect info leak. 2009.

[13] E. Gkta, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in Proceedings of

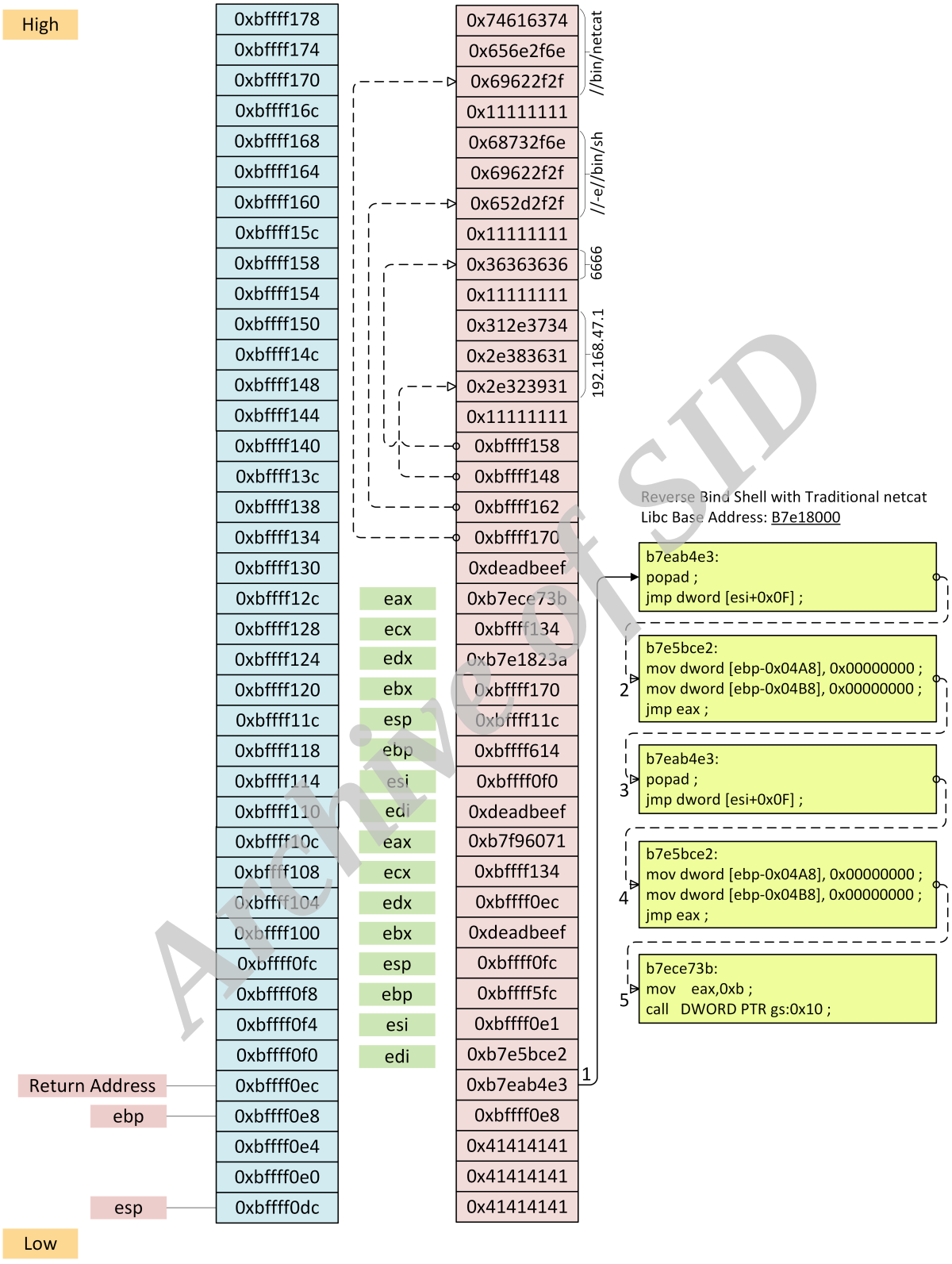

Figure 11. Reverse shell attack.

Table 5. A comparison between available shell-codes and tiny implemented ones.

	Tiny Method			Regular Methods			
Description	Type	Number of Gadgets	Shell-code Size (bytes) addr&data	Type	Number of Gadgets	Shell-code Size (bytes) addr&data	Reference
execve("/bin//sh", ["/bin//sh", NULL]) shellcode	JOP	2	36	JOP	11	216	[42]
execve("/bin//sh", ["/bin//sh", NULL]) shellcode	JOP	2	36	JOP	29	184	[8]
execve("/bin//sh", ["/bin//sh", NULL]) shellcode	JOP	2	36	JOP	22	224	[9]
execve("/bin//sh", ["/bin//sh", NULL]) shellcode	JOP	2	36	JOP	8	N/A	[28]
execve("/bin//sh", ["/bin//sh", NULL]) shellcode	JOP	2	36	JOP	9	N/A	[43]
exit(0);	JOP	3	36	JOP	4	52	[42]
exit(-1);	JOP	1	12	JOP	4	52	[42]
exit() with random value	JOP	1	4	JOP	4	52	[42]
kill all processes	JOP	3	36	JOP	9	160	[42]
kill all processes	JOP	3	36	JOP	10	N/A	[43]
killall5 shellcode	JOP	3	58	JOP	9	188	[42]
killall5 shellcode	JOP	3	58	JOP	9	N/A	[43]
PUSH() reboot	JOP	3	56	JOP	9	186	[42]
PUSH() reboot	JOP	3	56	JOP	9	N/A	[43]
set system time to 0	JOP	3	36	JOP	-	-	-
set system time to 0 and exit (-1)	JOP	3	48	JOP	12	212	[42]
set system time to 0 and exit (-1)	JOP	3	48	JOP	14	N/A	[43]
Reverse Bind Shell with Traditional netcat	JOP	5	140	JOP	4	52	-
Bind Shell with Traditional netcat	JOP	5	120	JOP	-	-	-
execve(/sbin/halt, /sbin/halt)	JOP	3	54	JOP	-	-	-
unlink "/etc/shadow"	JOP	3	52	JOP	-	-	-
chmod(/etc/shadow, 0666)	JOP	3	56	JOP	-	-	-

the 23rd USENIX conference on Security Symposium, 2014, pp. 417432.
[14] A.-A. Sadeghi, F. Aminmansour, and H. Shahriari, "Tiny Jump-oriented Programming Attack (A Class of Code Reuse Attacks)," in 12th International ISC Conference on Information Security and Cryptology (ISCISC), Guilan, Iran, 2015.
[15] ZadYree, "HT Editor 2.0.20 Buffer Overflow (ROP PoC)," 13-Nov-2012. [Online]. Available: http://www.exploit-db.com/exploits/22683/.
[16] A. Bacchelli, "Mining challenge 2013: Stack overflow," in The 10th Working Conference on Mining Software Repositories, 2013.
[17] G. Novark and E. D. Berger, "DieHarder: securing the heap," in Proceedings of the 17th ACM conference on Computer and communications security, 2010, pp. 573584.
[18] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 760770.
[19] S. M. Pike, B. W. Weide, and J. E. Hollingsworth, "Checkmate: cornering C++ dynamic memory errors with checked pointers," in ACM SIGCSE Bulletin, 2000, vol. 32, pp. 352356.
[20] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 CWE/SANS top 25 most dangerous software errors," Common Weakness Enumer., vol. 7515, 2011.
[21] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in Proceedings of the 15th ACM conference on Computer and communications security, 2008, pp. 1526.
[22] S. Andersen and V. Abella, Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. 2004.
[23] L. Le, "Payload already inside: datafire-use for ROP exploits," Black Hat USA, 2010.
[24] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "DROP: Detecting return-oriented programming malicious code," in Information Systems Security, Springer, 2009, pp. 163177.
[25] M. Polychronakis and A. D. Keromytis, "ROP payload detection using speculative code execution," in Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on, 2011, pp. 5865.
[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," ACM Sigplan Not., vol. 40, no. 6, pp. 190200, 2005.
[27] F. Yao, J. Chen, and G. Venkataramani, "JOP-alarm: Detecting jump-oriented programming-based anomalies in applications," in Computer Design (ICCD), 2013 IEEE 31st International Conference on, 2013, pp. 467470.
[28] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "SCRAP: Architecture for signature-based protection from code reuse attacks," in High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on, 2013, pp. 258269.
[29] Z. Huang, T. Zheng, Y. Shi, and A. Li, "A dynamic detection method against ROP and JOP," in Systems and Informatics (ICSAI), 2012 International Conference on, 2012, pp. 10721077.
[30] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing.," in USENIX Security, 2013, pp. 447462.
[31] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in Symposium on Network and Distributed System Security (NDSS), 2014.
[32] A.A. Sadeghi, F. Aminmansour, and H.R. Shahriari, "Tazhi: A novel technique for hunting trampoline gadgets of jump oriented programming (A class of code reuse attacks)," in Information Security and Cryptology (ISCISC), 2014 11th International ISC Conference on, 2014, pp. 2126.
[33] F. Aminmansour and H. R. Shahriari, "Patulous Code Reuse Attack: A novel code reuse attack on ARM architecture (A proof of concept on Android OS)," in 2015 12th International Iranian Society of Cryptology Conference on Information Security and Cryptology (ISCISC), 2015, pp. 104109.
[34] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in USENIX Security Symposium, 2014.
[35] H. Marco-Gisbert and I. Ripoll, On the Effectiveness of Full-ASLR on 64-bit Linux. DeepSeC, 2014.
[36] L. Davi and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in 23rd USENIX Security Symposium.
[37] A. Sotirov and M. Dowd, "Bypassing browser memory protections in Windows Vista," Blackhat USA, 2008.
[38] Mike: Operating Systems Development - Errors, Exceptions, Interruptions. (2008). Mike, "Operating Systems Development - Errors, Exceptions, Interruptions," 2008. [Online]. Available: http://www.brokenthorn.com/Resources/OSDev15.html.

[39] J. Salwan, "Shellcodes database." [Online]. Available: http://shell-storm.org/shellcode/.
[40] Zad, "87289: HT Editor Filename Handling Overflow," 2012. [Online]. Available: http://osvdb.org/show/osvdb/87289.
[41] V. Ramachandran, "Demystifying the Execve Shellcode (Stack Method)," 2013. [Online]. Available: http://hackoftheday.securitytube.net/2013/04/demystifying-execve-shellcode-stack.html.
[42] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin, "Automatic construction of jump-oriented programming shellcode (on the x86)," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, 2011, pp. 2029.
[43] P. Chen, X. Xing, H. Han, B. Mao, and L. Xie, "Efficient detection of the return-oriented programming malicious code," in Information Systems Security, Springer, 2011, pp. 140155.
[44] Aminmansour, Farzane, and Hamid Reza Shahriari. "Aggrandizing the beast's limbs: patulous code reuse attack on ARM architecture." The ISC International Journal of Information Security 8, no. 1, 2016, pp. 39-52.

Aliakbar Sadeghi received the M.S. degree in information security from department of computer engineering and information technology, Amirkabir University of Technology in 2015. He is currently an information security researcher in the same department. His M.S. dissertation was in the field of exploit development entitled "Behavior-based detection of jump-oriented programming shell code". His research interests include exploit development and especially software vulnerability analysis.

Farzane Aminmansour is graduated from the department of computer engineering and information technology at Amirkabir University of Technology with a master's degree in 2016. She received her bachelor's degree in information technology from University of Isfahan in 2013. Her research interests include information security, especially low-level system and software security, operating system security, mobile system and application security, and malware analysis.

HamidReza Shahriari is currently an assistant professor at the department of computer engineering and information technology at Amirkabir University of Technology. He received his Ph.D. in computer engineering from Sharif University of Technology in 2007. His research interests include information security, especially software vulnerability analysis, security in e-commerce, trust and reputation models, and database security.

Appendix

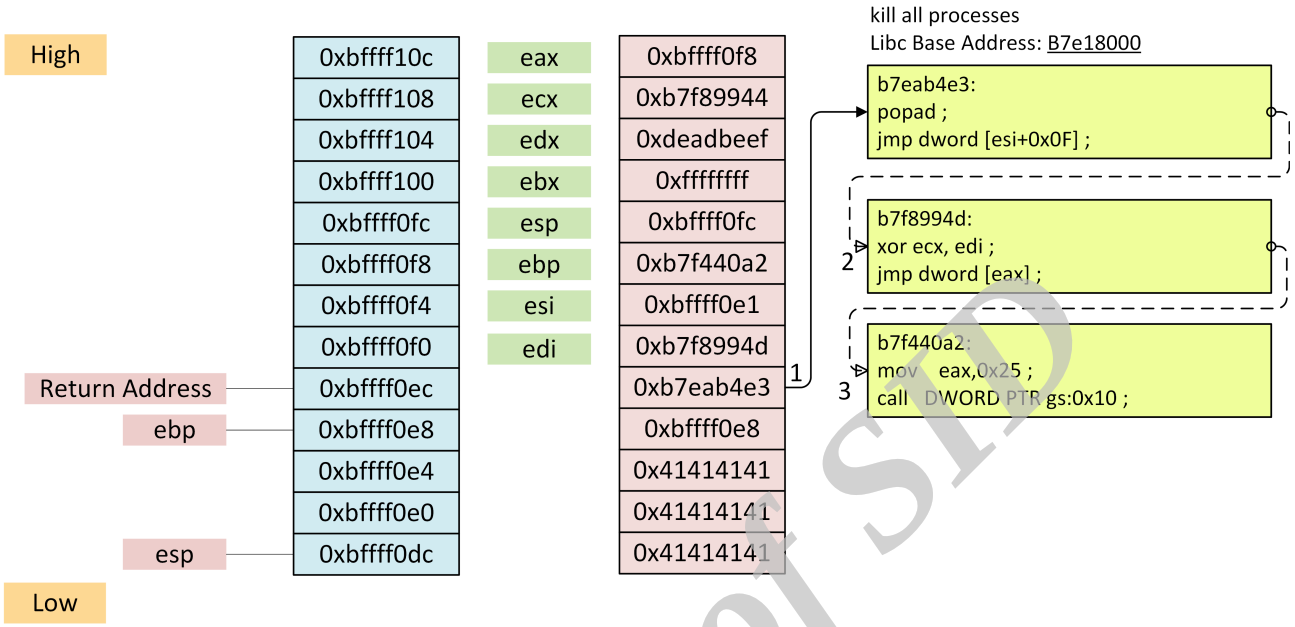

Figure 12. Kill all process attack

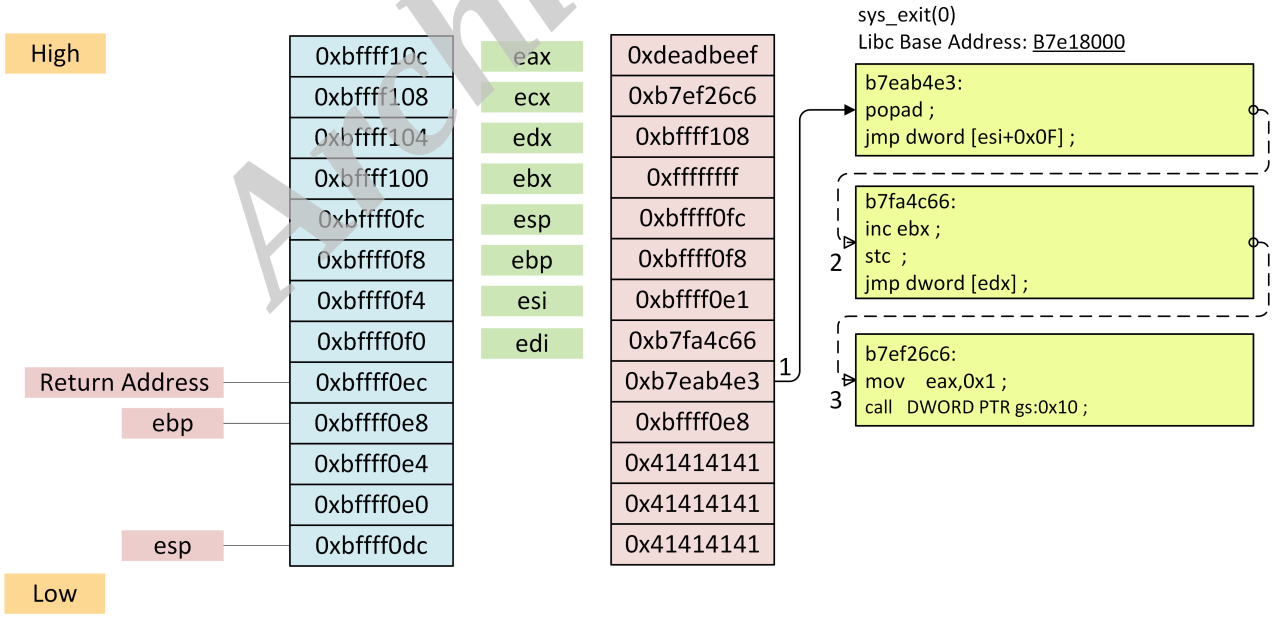

Figure 13. Sys_exit (0)

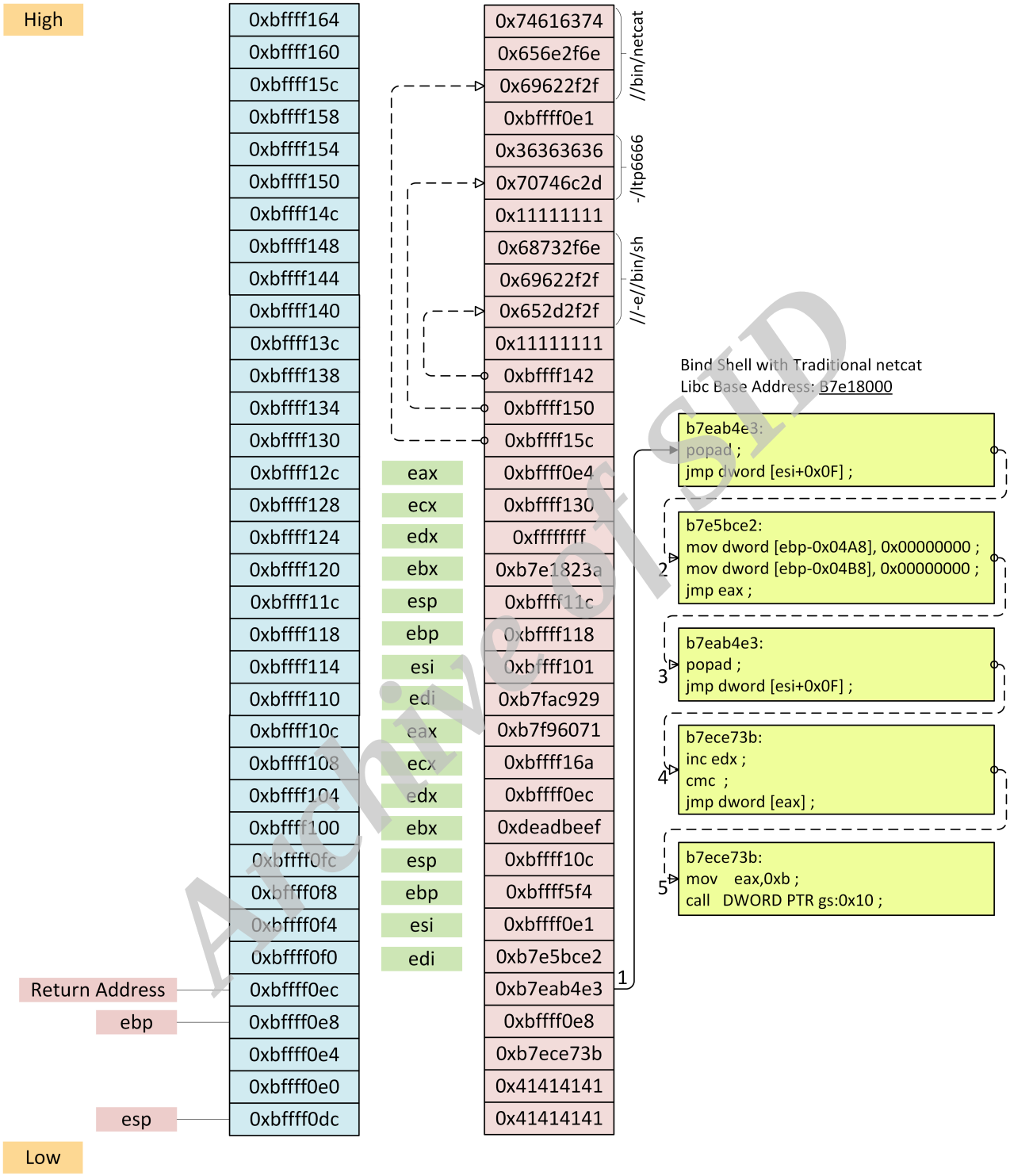

Figure 14. Bind shell with traditional Netcat

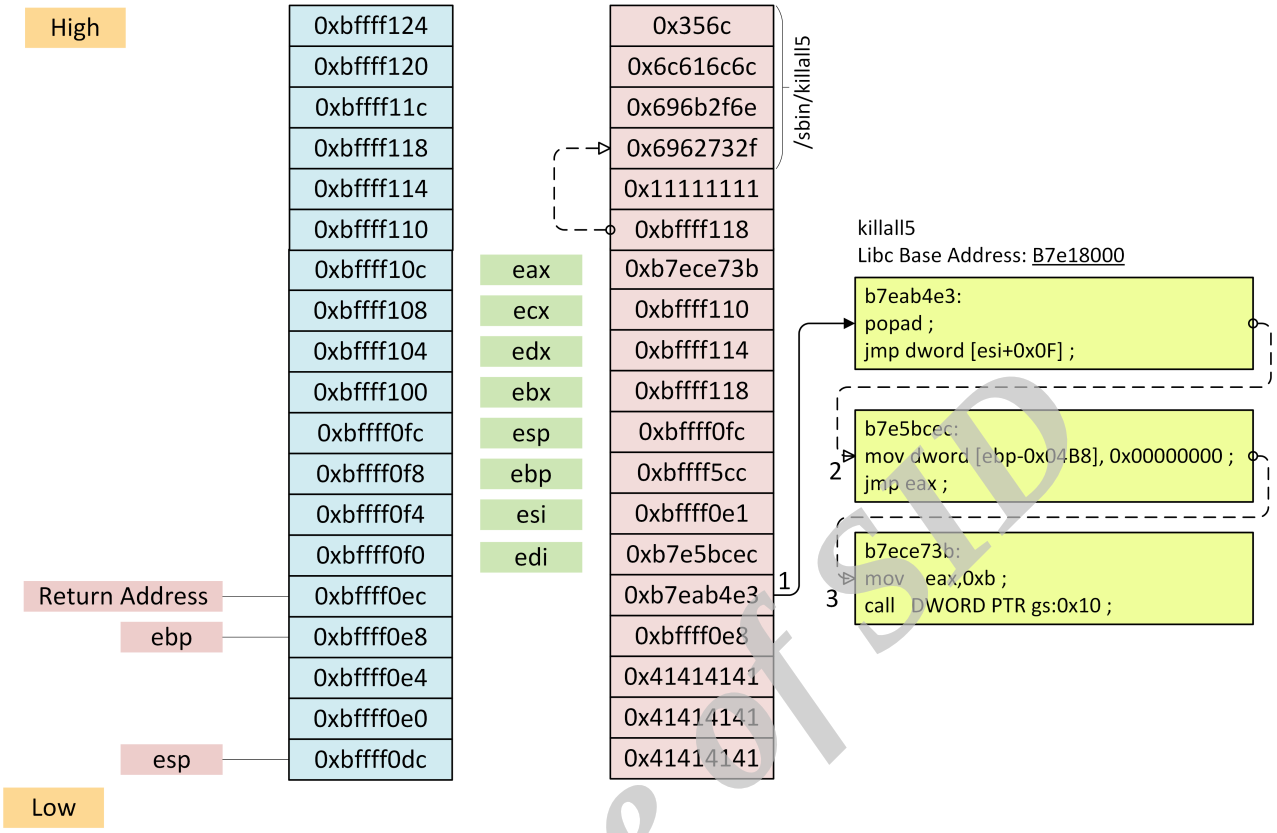

Figure 15. Killall5 attack

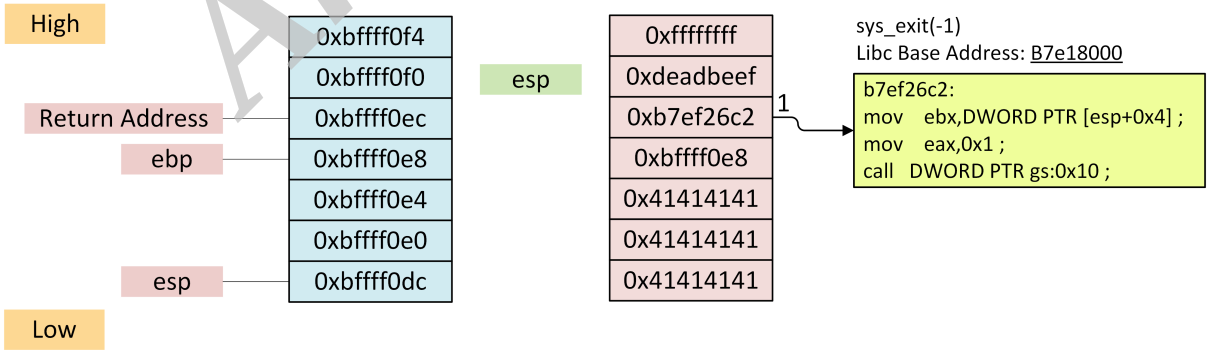

Figure 16. Sys_exit (-1)

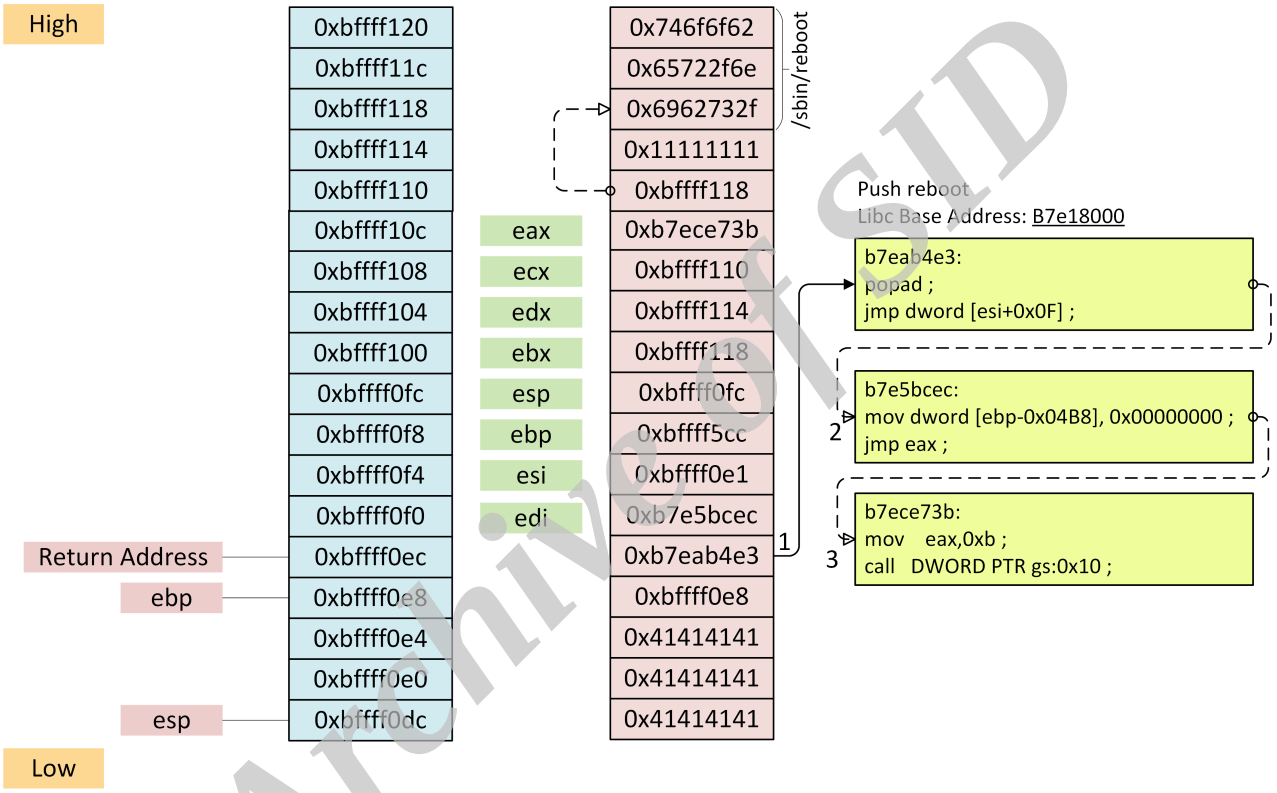

Figure 17. Push reboot attack

The ISC Int'l Journal of Information Security
ISeCure

January 2017, Volume 9, Number 1 (p. 97)
http://www.isecure-journal.org

Persian Abstract

فرانکشتاین کوتوله هنوز در حافظه موجود است: حمله بازاستفاده از کد کوچک

علی‌اکبر صادقی[1]، فرزانه امین‌منصور[1] و حمیدرضا شهریاری[1]

[1] دانشکده مهندسی و فناوری اطلاعات، دانشگاه صنعتی امیرکبیر، تهران، ایران

حملات باز استفاده از کد مانند حمله‌ی برنامه‌نویسی بازگشت‌گرا و حمله‌ی برنامه‌نویسی پرش‌گرا در میان حمله‌کنندگان بسیار محبوب می‌باشند. تعداد بسیار زیادی روش دفاعی عملی و غیرعملی به منظور جلوگیری از این حملات ارائه‌شده‌اند. این سیستم‌های تشخیص در سربار محاسباتی و کارایی، نیازمندی به سورس کد برنامه، نرخ تشخیص و نیازمندی‌های پیاده‌سازی با هم متفاوت می‌باشند. یکی از رفتارهای قابل استفاده در این سیستم‌های تشخیص، استفاده از طول زنجیره گجت‌های اجرا شونده حمله می‌باشد. استفاده از نرخ آستانه تعداد گجت‌های اجرا شونده باعث ایجاد نرخ مثبت کاذب (False Positive) و منفی کاذب (False Negative) در این سیستم‌های تشخیص می‌شود. نوآوری اصلی این مقاله ارائه یک روش هوشمندانه در طراحی حملات باز استفاده از کد می‌باشد که ما این روش را Tiny Code Reuse Attack نام‌گذاری کرده‌ایم. این روش ناکارآمد بودن روش‌های تشخیص مبتنی بر نرخ آستانه بر روی تعداد گجت‌های اجرا شونده را نشان می‌دهد. همچنین با حداقل مفروضات نماش داده‌ایم که Tiny-CRA گجت‌های اجرا شونده حمله را به حداقل می‌رساند. بنابراین سیستم تشخیص توانایی تشخیص تمایز بین اجرای نرمال برنامه و اجرای گجت‌های حمله را ندارد. به منظور انجام این کار ما گجت‌های پایه و گجت‌های مفید موجود در کتابخانه C را استخراج کرده‌ایم. همچنین به منظور نشان داده کارایی این روش ما ۹ کدپوسته متفاوت را طراحی و با استفاده از یک حمله واقعی سر ریز بافر در نرم‌افزار HT Editor 2.0.20 پیاده‌سازی کرده‌ایم.

واژه‌های کلیدی: امنیت نرم‌افزار، حملات باز استفاده از کد، حمله برنامه‌نویسی پرش‌گرا، حمله برنامه‌نویسی پرش‌گرا کوچک، گجت Kernel Trapper.