JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 27, 869-890 (2011)

A Robust Kernel-Based Solution to Control-Hijacking Buffer Overflow Attacks*

LI-HAN CHEN, FU-HAU HSU, CHENG-HSIEN HUANG, CHIH-WEN OU, CHIA-JUN LIN AND SZU-CHI LIU
Department of Computer Science and Information Engineering
National Central University
Taoyuan, 320 Taiwan

In this paper, we propose a robust kernel-based solution, called AURORA, to a ubiquitous security problem – control-hijacking Buffer Overflow Attacks (BOAs). AURORA utilizes either the addresses of the buffers storing input strings or signatures to detect and block control-hijacking BOA strings in the kernel, including zero-day ones. Although AURORA detects some types of BOAs through signatures, AURORA does not need to create any new signature for new attack instances after its installation because AURORA's signatures are created based on commonality of control-hijacking BOAs. Moreover, even a process is under a BOA, AURORA allows it to continue its execution or to be terminated gracefully without the cost of process idleness or repeated process crashes. Thus, AURORA is robust to control-hijacking BOAs. AURORA does not need to modify the source code of any application programs. Furthermore, AURORA is compatible with existing operating systems and application programs; hence, AURORA could work with other protection mechanisms to provide an extra layer of protection. Our experimental results show that with less than 1% overhead and negligible false positives, AURORA can accurately block various control-hijacking BOAs.

Keywords: buffer overflow attack, stack smashing attack, return-into-libc attack, AURORA, control hijacking BOA

1. INTRODUCTION

Since they first appeared in computers about half a century ago, *Buffer Overflow Attacks* (BOAs) have continuously been one of the most dangerous and common security threats to computer and network systems. According to US-CERT [1], there were tens of new BOAs appeared each month in 2008. A control-hijacking BOA is launched through overwriting control sensitive data (such as return addresses, function pointers, GOT entries and the `jumpbuf`) with a new address, called a *deviation address*, to transfer the execution flow of a program into the code injected/chosen by attackers. Stack smashing BOAs and return-into-libc BOAs [9, 14, 39] are two most common control-hijacking BOA types, especially the former. This paper proposes a solution, called AURORA, to solve stack smashing BOAs and return-into-libc attacks, including zero-day ones.

Even though there are many secure solutions to defend a system against BOAs, they are not inexpensive. Usually the cost is either process idleness or repeated process crashes. Attacked processes are idle because the processes continue waiting for attackers' input which has already been blocked by a protection mechanism. Attacked processes

Received October 19, 2009; revised January 26, 2010; accepted March 3, 2010.
Communicated by Chin-Laung Lei.
* This paper was partially supported by the National Science Council open source project and Advanced Communication Laboratory in Nation Central University. The number of the project is NSC 97-2218-E-008-006.

crash because the response taken by some BOA solutions to detected BOAs is to crash the related processes. Moreover, a BOA usually destroys the address space of an attacked process, which could also result in the crash of the process. Therefore, an attacker can continue crashing processes on a host by repeatedly sending the host the same attack strings, metamorphic strings, or meaningless long strings. Due to the above reasons, one of the design goals of AURORA is to make it robust to the crash and idleness problem. AURORA attains this aim by first blocking attack traffic in the operating system kernel before attack traffic pollutes the address space of an attacked process. Then by sending "socket close'' or "end-of-file'' to the attacked process as a response to the related service request and closing related sockets/files, AURORA allows the attacked process to continue its execution based on its original algorithm.

Modern OSes use a system call, such as `read` or `recv`, to retrieve network data into a user mode buffer called an *input buffer* hereafter. One of the parameters of these system calls is the base address of an input buffer, called *input buffer address*. AURORA utilizes both the input buffer addresses and signatures to detect and block related attack traffic in the kernel. Hence, AURORA is a kernel-based solution to stack smashing BOAs and return-into-`libc` attacks. From the input buffer address of a `read` or `recv` system call, the kernel can realize where external input will be stored in the address space of the related process. Moreover, because the kernel also knows how many bytes will be copied into the input buffer, the kernel can accurately comprehend what part of the user address space of the process will be changed by the system call. Based on the above information and the stack frame layout, the kernel can know exactly whether a `read` or `recv` system call will directly overwrite the return address field or caller `ebp` field of the stack frame of any active function when the system call is executed.

The above solution works well when an overflow directly overwrites the return address field or caller `ebp` field adjacent to an input buffer. But an overflow may overwrite a function pointer adjacent to an input buffer or an overflow may not happen through an input buffer directly. The latter occurs when the content of an input buffer is copied to a different buffer and an overflow occurs through that non-input buffer. AURORA uses signatures to detect the above two types of BOAs inside the kernel and blocks unsafe input there before it is copied into the target area.

Most signature-based solutions require a large signature database due to the need to craft a signature for every different attack. Unlike these solutions, AURORA only uses fixed number of signatures to detect stack smashing BOAs and return-into-libc attacks, including zero-day ones. Thus, it is a signature-generation free solution and no extra efforts are required to maintain AURORA's signatures after its installation. On the contrary, most signature-based solutions need to continuously update their signature databases because new attacks appear on a daily basis. Even though many *Automatic Signature Generation Mechanisms* (ASGM) have been proposed to reduce manpower, their effectiveness is decided by the following conditions. First, an ASGM must know the input formats of protected processes. Second, attack strings must follow the same input formats. Third, an ASGM must obtain attack string samples in advance. However, the input formats are not available for a lot of freeware and many programs even do not have well-defined input formats. Moreover, because attack strings are collected through a honeypot or dump files, if an attacker didn't attack related honeypots, corresponding signatures could not be generated in time. Furthermore, buggy programs crash often even when there is no attack; hence, the corresponding dump file may result in the creation of wrong signatures. Finally an ASGM can't create signatures from meaningless strings whose only purpose is to crash processes that have buffer overflow vulnerabilities.

AURORA creates signatures based on some common properties of the *attack payload elements*. Hence, they are suitable for various control-hijacking BOAs, including zero day ones. Attack payload elements are those who will be injected into the address space of an attacked process when a control-hijacking BOA is launched. These elements are collectively called *attack payloads*. For a stack smashing BOA, an attack payload consists of a deviation address, shell code, and an optional **NOP** sled. For a return-into-libc attack, an attack payload comprises a deviation address and optional parameters. The most popular return-into-libc attacks usually use function `system()` or system call `execve()` to execute a program, such as /`bin`/`sh`; hence, the most commonly seen return-into-libc attack payloads consist of a deviation address, an address parameter which points to a program name string, and the program name string.

Among the above attack payload elements, **NOP** sleds, deviation addresses, and address parameters are three of the most important ones. Therefore, one of AURORA's most important jobs is to accurately filter out these three elements from input strings. According to [46], in an IA32 host, except privileged instructions, there are about 52 one-byte instructions which could be used to constitute a **NOP** sled. 0×90 is not the only one-byte instruction that could be used to create a **NOP** sled. Hence, a sequence of bytes forming from these one-byte instructions is deemed as a **NOP** sled if its length is larger than a threshold defined by AURORA. For the deviation address and address parameter part, both our analyses and experiments show that the value of a deviation address or the value of an address parameter usually is close to the related input buffer address if the input buffer is located at the stack segment of the attacked process. But if the input buffer is located at the data segment or heap segment, the above values are close to the caller `ebp` field address of the stack frame of the function that reads data into the input buffer. The details will be discussed in later section.

AURORA divides network hosts into three types, *normal hosts*, *attack hosts*, and *crash hosts*, based on their behavior observed by AURORA. Hosts that have been found sending control-hijacking BOA strings to a computer several times are deemed as attack hosts by the attacked computer. Attack hosts that continuously attack a computer are treated as crash hosts by the computer. All other hosts are normal hosts. AURORA's signatures can be divided into two types, *normal signatures* and *attack signatures*. Normal signatures are used to examine input originating from normal hosts. Attack signatures are used to check input stemming from attack hosts. Hence, depending on the origin type of an input string, AURORA automatically chooses appropriate signatures to check the content of the string.

Because normal signatures are applied to normal hosts, they are designed to have both low false positives and low false negatives. But signatures created under this restriction usually are relatively more easily to be bypassed. AURORA solves this dilemma by increasing the failure possibility of an attack if the attack tries to bypass AURORA's signatures. An unsuccessful attack will result in the attacked process' crash, which in turn will trigger AURORA to mark the malicious host as an attack host and use stricter signatures, *i.e.* attack signatures, to check its future input. The attack signatures have higher false positives but are difficult to be bypassed. Thus, AURORA uses them to intercept attack traffic in the kernel to avoid crash of an attacked process. AURORA directly drops traffic from crash hosts due to their notorious records. For a multiuser host, the above security policy may forbid other innocent users of a crash host to use AURORA-protected hosts. However, with or without AURORA, a vicious user in a multiuser host still could prohibit other users in the host from using any services in any hosts by adopting the following steps – monitoring network traffic of his/her host and sending RESET or SYN/ACK packets to disconnect a TCP connection. Thus, AURORA does not make it easier for attackers to launch a DoS/DDoS attack against hosts protected by it.

Whenever AURORA detects an attack payload or a process crash, before forking a new process/thread to serve next request, AURORA dynamically creates an environment variable with randomly chosen length and inserts this variable into the address space of process which is going to be created. Because every character of environment variables will be stored in a process's user mode stack, this variable will change the positions of all objects that will be stored subsequently in the stack segment, such as return addresses, local variables, and formal parameters. This approach, called a *lightweight ASLR*, can prevent attackers from coordinating multiple attacks to infer a suitable attack payload.

AURORA does not change the structure of the underlying operating system; hence, it does not have any compatible problems with any application programs. It can also work with other BOA protection mechanisms to provide an extra layer of protection. In addition, because AURORA is an OS-based solution to BOAs, it does not need to modify the source code files or executable files of any application program. We implemented AURORA in a Linux host. Experimental results show that with negligible false positives and less than 1% overhead, AURORA can accurately block various control-hijacking BOAs. The rest of this paper is organized as follows. Section 2 describes other BOA-related research. Section 3 gives a detailed description of AURORA. Section 4 analyzes the experimental results of AURORA implemented in a Linux platform. Section 5 addresses the conclusion.

2. RELATED WORK

Due to the serious destructive power and abundant mutants that BOAs have, BOAs are one of the most obstinate problems of system and software security and have attracted plenty of researchers to work on this notorious problem. Along with tremendous research efforts, more and more promising solutions have been proposed. In the rest of this section, we will make a brief introduction to them.

Several techniques solve BOAs by changing the layout or number or property of stacks. StackGuard [2], one of the most famous and important buffer overflow-related research works, inserts a canary word before the return address of the stack frame of every active function. By checking the integrity of the canary word right before transferring control back to the caller function, StackGuard prevents attackers from hijacking the execution of an process through overwriting a return address.

RAD [18] protects return addresses by storing another copy of return addresses in a well-protected area, called RAR. By checking the consistence of return addresses in RAR with their counterparts in the stack, RAD protects programs against stack smashing BOAs. Both RAD and StackGuard utilize compilers to automatically add integrity or consistence check code into the programs protected by them. However, in order to provide the protection, like all other compiler-based solutions, both methods require source code.

Solar designer [14] utilizes a non-executable stack to solve stack smashing BOAs. However, this approach has compatible problems with nested functions and the `sigreturn` system call in Linux. Younan *et al*. [3] classify local variables into several categories according to their types. By storing each different class of variables at a different stack, their method prevents attackers from taking over program execution by overflowing a datum buffer and its adjacent area. Even though the extra stacks provide good protection to the stack segment of a process, they consume more memory resource and introduce a compatible problem to stack-layout-sensitive programs.

One of the most promising solutions to BOAs is Address Space Layout Randomization (ASLR/PaX). OSes released recently, such as the latest versions of Linux, BSD, and Windows Vista, include this feature as one of their options. The fundamental concept of this approach is to randomize the location of different process units, such as code, data, and various tables; hence this approach makes it difficult for the attackers to transfer program execution into the code chosen by them. Several implementation strategies [7, 8, 48, 49, 57] with different granularity of randomness were proposed. The performance overhead of this kind of solutions usually is low. However, internal fragments, process crash problems for commercial sites, and compatible problems for non-relocatable programs are unwanted side-effects of ASLR.

Some solutions utilize cryptography to solve BOAs. However, these kinds of solutions to some extent suffer from compatibility problems. Instructions or data that are not encrypted accordingly usually can not work with those that have been encrypted. Shared libraries and shared data are the major sources of this problem. Programs recompiled by PointGuard [43] encrypt pointer values to protect pointer-related data from pointer-related attacks. Kc *et al*. [10] and Barrantes *et al*. [50] use an approach commonly known as Instruction Set Randomization (ISR) to solve code injection style BOAs. ISR exploits a key to encrypt the machine code of an executable and decrypt the code right before they are going to be executed. Hence, without knowing the right key, attackers are unable to inject correct encrypted code for execution. However, in order to make this solution more efficient, hardware support is unavoidable. But the price of adding new hardware usually is not inexpensive. In addition, Sovarel *et al*. [13] proposed an incremental key-breaking attack approach to invalidate ISR.

Since the fundamental cause of BOAs is a lack of size checking between input strings and the corresponding input buffers, array bound checking uses compilers to automatically add array bound checking code into programs compiled by them. Several approaches [40-42, 51] have been developed to implement automatic array bound checking. However, currently these techniques still suffer from high performance overhead and changes to pointer semantic. The overhead is caused by the need to check the correctness of every memory access.

Static source code analysis techniques [53-56] automatically analyze source code of application programs to detect potential buffer overflow vulnerabilities. These schemes usually suffer from high false alarms and scalability problems.

In their Nebula project, Hsu *et al*. [5] proposed a network-based solution to detect BOA strings targeting at Linux hosts on the fly. Because the stack segment of a Linux process grows downward from address `0xC0000000` and usually the size of a Linux process's stack segment is less than 8MB, they use the address range `0xC0000000` ~ (`0xC0000000-8M`) as the criterion to detect the deviation address substrings hidden in an attack string. Toth and Kruegel [45] disassembled a packet content to see whether it contains a sequence of 30 or more instructions to detect code injection style BOAs. Dalton *et al*. [47] used DIFT to prevent attackers from overwriting pointers in the application or OS with no false positives.

Buttercup [58] uses addresses as hints to detect buffer overflow attacks. Instead of using a generic address pattern for all buffer overflow attacks, for each individual attack string, they need to study the specific vulnerable program and its buggy overflowable functions to drive a range of possible address that could be used to launch a successful attack. Later on this address range is used as a signature of the specific attack string; therefore, if any word of a packet's payload could be interpreted as an address within this range, the packet is classified as an attack packet. This method simplifies the signatures of known attacks; thus, improves the performance of signature matching. However, current Buttercup version can not handle unknown buffer overflow attacks.

The majority of worms utilize buffer overflow vulnerabilities to proliferate themselves; hence, worm signatures could also be used to block BOAs. In this paper, we only discuss worm-related works that automatically generate worm signatures. Earlybird [63], autograph [64], and Honeycomb [65] generate worm signatures based on detected attack packets. A common long substring appearing in all detected attack packets related to the same worm is deemed as the signature of the worm. Because polymorphic worms modify themselves as they spread in the Internet, signatures created by the above works could not correctly detect them.

Polygraph [66] generates a worm signature based on the invariant parts of various network traffic associated with the same polymorphic worm; thus, it provide more robust protection to hosts against polymorphic worms. However, this work suffers from non-trivial false positives and false negatives. Nemean [67] incorporated protocol semantics into their signature generation algorithm to improve both the coverage and the precision of its signatures. However, by launching a noise injection attack [71], an attacker can mislead the above approaches to generate wrong signatures.

Vigilante [68] and TainCheck [69] utilize dynamic dataflow/taint analysis to capture various worms; however, their performance overhead is non-trivial. Sigfree [59] uses code abstraction to examine whether a network input stream contain a meaningful code sequence; therefore, without the need to maintain a signature database, Sigfree can block various code-injection style BOAs. But Sigfree can not capture return-into-libc attacks.

Based on crash dumps caused by unsuccessful attacks, COVERS [70] utilizes correlation and input context identification to generate signatures for a broad range of memory error exploits. Because the statuses of CPU registers, the stack of the related crash process, and network traffic format of the process are used in their signature generation process, COVERS can create signatures with less false positives and false negatives comparing with previous work. ShieldGen [52] can also automatically generate signatures for a wide range of attacks if a zero-day attack instance is provided. With the knowledge of the data format and protocol context of a protected process, ShieldGen generates new potential attack instances. Then by using its zero-day detector, ShiedGen checks whether the detector can block these newly created attack instances. Finally based on the feedback of the detector, ShieldGen creates signatures for the corresponding zero-day attack.

Even though ShieldGen can effectively reduce the number of false positives; however, it still creates non-trivial false negatives.

3. ATTACK DETECTION MECHANISMS OF AURORA

AURORA detects stack smashing BOAs and return-into-libc attacks either by observing the memory area influenced by a `read` or `recv` system call or by signatures. In this section we explain these two approaches in detail.

3.1 Memory Area Observation Method (MAOM)

As described in section 1, when a process executes a `read` or `recv` system call, inside the kernel, from the address of the input buffer and the length of the input string, the kernel can obtain the entire region that will be changed by the system call. Hence, through the `ebp` register value stored in the kernel mode stack of the process and the caller `ebp` field of the stack frame of every active function, the kernel can fully understand whether the `read` or `recv` system call will directly overwrite the return address or the caller `ebp` field of the stack frame of any active function of the process. By intercepting this unsafe read, AURORA can prevent attackers from transferring process execution flow into the code injected/chosen by them. This approach, called *Memory Area Observation Method* (MAOM), is a major component of AURORA. In addition, because a function obtains its first parameter from the memory location with address `ebp + 8`, in order to provide the intended parameter to the function chosen by a return-into-libc attack, attackers inevitably need to overwrite the caller `ebp` field of the topmost stack frame. Therefore, even if an attacker launches a return-into-libc attack through overflowing a function pointer, AURORA still can use MAOM to detect the attack.

3.2 Properties of Control-Hijacking BOAs

AURORA uses MAOM to detect a BOA which directly overflows the caller `ebp` field or return address of a stack frame through an input buffer. However, a control-hijacking BOA could be launched through a non-input buffer or a function pointer; hence, they can not be detected by MAOM. AURORA uses signatures to solve this limitation. This subsection analyzes the properties of different control-hijacking BOAs and their corresponding attack strings. Based on the results, AURORA creates signatures to detect control-hijacking BOAs that can not be detected by MAOM.

3.2.1 Overflowable buffers and attack payloads

A BOA uses a memory buffer as a stepping stone to overflow adjacent control sensitive data to launch its attack. We call the above stepping stone buffer an *overflowable buffer*. For a control-hijacking BOA, the related overflowable buffer usually is located at the stack segment of a process, because the most frequently used control sensitive data, such as return addresses and local function pointers, are stored there. However, if a vulnerable program uses a global function pointer to invoke functions, then an overflowable buffer may also be located at the data segment. An overflowable buffer could be an input buffer. But it may also be a non-input buffer, because external input may be read into an input buffer first and then be copied to an overflowable buffer. If an overflow occurs in an input buffer, it is called a *direct overflow*. Otherwise, it is called an *indirect overflow*. MAOM could not detect indirect overflows and direct overflows that overwrite a function pointer. Hence, signatures are used to solve this limitation. In order to create high quality signatures to filter out attack strings, inevitably we need to understand the properties of these strings.

To launch a successful control-hijacking BOA, attackers must inject several different elements, collectively called *attack payloads*, into the address space of the attacked process. For a stack smashing BOA, an attack payload consists of a deviation address, shell code, and an optional **NOP** sled. For a return-into-libc attack, an attack payload comprises a deviation address and optional parameters. For both the above BOA types, usually the whole attack payload associated with a BOA is written into an overflowable buffer. However, some elements of an attack payload, such as the **NOP** sled and shell code, could be stored in a different buffer. But to make this arrangement, the attacked process must have the related vulnerabilities which allow an attacker to manipulate the contents of at least two different buffers simultaneously without being interfered by the attacked process. Based on the properties of attack payload elements and the origins of attack strings, AURORA designs distinct signatures to handle BOAs.

AURORA designs its signatures based on the *indispensable properties* and *important properties* of attack payload elements. An indispensible property is a property that is critical to a successful attack. Without it, an attack could not succeed. An important property is a property that could increase the chance of obtaining a successful attack. However, without it, an attacker still can launch a successful attack if the attacker is lucky enough. Because AURORA designs its signatures based on the indispensable properties and important properties of control-hijacking BOAs, not some specific cases, the signatures are suitable for various BOAs, including zero-day ones. According to the above principles, AURORA creates two different signatures, *normal signatures* and *attack signatures*, to examine input originating from distinct hosts which contain *normal hosts*, *attack hosts*, and *crash hosts*. Normal signatures are used to filter input originating from normal hosts. Attack signatures are used to check input stemming from attack hosts. Hence, depending on the origin type of an input string, AURORA automatically chooses appropriate signatures to check its content.

In the rest of this subsection, we discuss the structure of both a stack smashing BOA string and a return-into-libc attack string so that we can get the indispensable properties and important properties of these two types of attack strings.

3.2.2 Properties of a stack smashing BOA string

The attack payload of a stack smashing BOA must contain injected code and a deviation address; hence, the indispensable properties of the attack payload of a stack smashing BOA are (1) code and (2) the deviation address. However, in order to launch a successful stack smashing BOA, the deviation address must be written into the right place with the right value. The right place is a memory area containing control sensitive data, such as a return address. The right value is the entry point address of the injected shell code. However, this entry point address is not predictable due to the following reasons. In Unix OS family, when a process is created, the system copies all environment variables and the command line string into the initial area of the stack segment of the process first. Then the stack frame of each active function is stored in the area right after the above data. Hence, varying the length of the environment variables or the command line string also adjusts the locations of the stack frames of all active functions and all stack frame elements, such as return addresses and local variables. We call the above phenomenon *push-effect*. Because an attacker could not know the content of the environment variables and the command line string of an attacked process, she/he can not know the exact location into which her/his code will be injected, even she/he has a local copy of the attacked program.

A long **NOP** sled is used to mitigate the above problem. A **NOP** sled consists of a sequence of one-byte non-privileged instructions and is located right before the injected code. Even though a **NOP** sled is usually constituted from 0 × 90 and 0 × 41 these two one-byte instructions, in an IA32 host there are about other 50 one-type instructions [46] which could be used to create a **NOP** sled. Table 1 shows the lengths of the **NOP** sleds of several released attack strings. All of them are more than 800.

Table 1. NOP sled lengths of various programs with buffer overflow vulnerabilities. '<' in the "NOP sled length'' column means the length must deduct the length of shell code which is usually less than 200 bytes.

Program	CVE	NOP sled length
3proxy 0.5.3g	CVE-2007-2031	1800
mbsebbs 0.70.0	CVE-2007-0368	4072
Snort	CVE-2005-3252	<1069
MIT Kerberos	CVE-2007-0957	900
Borland InterBase	CVE-2007-5243	<1024
Borland InterBase	CVE-2007-5244	<1056
Samba	CVE-2007-2446	1024

After detecting an attack payload or the crash of an attack process, lightweight ASLR is used to create the next new process. AURORA implements lightweight ASLR by inserting an environment variable with a randomly chosen length to the new born process. This action not only can easily change the addresses of all objects, such as return addresses, stored in the stack segment but also could force attackers to use a longer **NOP** sled if they want to improve the success chance of their attacks. According to the definition (section 3.2.1), a **NOP** sled is an important property of stack smashing BOAs.

3.2.3 Properties of a return-into-libc attack string

In a return-into-libc attack, attackers utilize a function of the attacked process to carry out the attack. Therefore, in a return-into-libc attack string, the address of a function is required. The function usually is a library function. However, if the code segment of an attacked process contains a suitable function for attackers, they will also use that function. In addition, research [44, 74] shows that the most frequently used functions in return-into-libc attacks are `system()` and `exec()`. Both these functions use the address of a program name string, such as /bin/sh, as their first parameter; hence, the most commonly seen return-into-libc attack payloads consist of a deviation address, an address parameter which points to a string that is the name of a program, and the program name string. Therefore, the indispensable properties of the attack payload of most common return-into-libc attacks are (1) the address of a function and (2) the address of a string.

In order to have a successful return-into-libc attack, attackers must know both the address of the function carrying out the attack and the address of a program name string. However, both of them are not easily to be obtained from a remote site, especially the latter due to the push-effect phenomenon. In addition, unlike stack smashing BOAs which could use **NOP** sleds to increase success possibility, a return-into-libc attack does not have other efficient tricks to increase its success possibility. We believe this is one of the reasons why even though the majority of stack smashing BOAs could be transferred to return-into-libc attacks, in the wild return-into-libc attacks are not common. In other words, a single return-into-libc attack is very likely to crash the attacked process. Hence, the indispensable properties of a return-into-libc attack are only used in AURORA's attack signatures to examine input coming from attack hosts which have bad record in attacking or crashing processes running in the hosts protected by AURORA.

3.3 AURORA Signatures

As mentioned in section 1, normal signatures and attack signatures are used to examine traffic originating from normal hosts and attack hosts respectively. Input coming from crash hosts is completely blocked by AURORA due to their bad records. This subsection gives a detailed description of these signatures and the approach used by AURORA to classify hosts.

AURORA deems each host as a normal host by default. After the input coming from a host with IP address x is detected to contain an attack payload or crashes a process, then a counter, x.counter, is assigned to that host. The initial value of x.counter is set to 1. Later on if input from the host is detected to contain an attack payload or crashes a process, and then its counter will be increased by one. When the counter is greater than value *attack_threshold*, the host is marked as an attack host. When the counter is greater than value *crash_threshold*, the host is deemed as a crash host. *attack_threshold* is smaller than *crash_threshold*. Based on their needs, system administrators can use these two thresholds to control system availability to suspected attack hosts. Because experimental results show that AURORA can accurately detect attack strings, we think low threshold values, such as 3 for *attack_threshold* and 4 for *crash_threshold*, are enough to satisfy most situations. AURORA uses lightweight ASLR (section 1) to create the next new process whenever the counter of a host increases. Because lightweight ASLR can vary most address-related attack payload elements, attackers can not use information collected from previous attacks to improve the success possibility of subsequent attacks.

Normal signatures have the following special features. First, normal signatures include important properties of stack smashing BOAs. Hence, carefully crafted attack strings may bypass AURORA's detection. However, these attack strings are very likely to crash related processes, which in turn will trigger AURORA to use stricter signatures, *i.e.* attack signatures, to examine traffic from the same resources or even to completely block traffic from them. Second, they do not filter out return-into-libc attack payloads from input. As discussed in the previous subsection, a host is unlikely to compromise an AUR-ORA-protected host through only several return-into-libc attacks. In other words, the host's attack strings are highly likely to crash the attacked processes, which in turn will trigger AURORA to mark it as an attack host and only use indispensable properties of return-into-libc attacks to filter traffic coming from that host.

Based on whether the input buffer of an input string is a stack buffer or a non-stack buffer, AURORA's normal signatures can be classified as the following two types.

- **Indirect Stack Input Buffer Signature**
  - The input buffer of an input string is a stack buffer.
  - A substring of the above string could be interpreted as a **NOP** sled.
  - A substring of the above string could be interpreted as a deviation addresses.
- **Indirect Non-stack Input Buffer Signature**
  - The input buffer of an input string is a non-stack buffer.
  - A substring of the above string could be interpreted as a **NOP** sled.
  - A substring of the above string could be interpreted as a deviation addresses.

NOP sleds and deviation addresses will be discussed in the next subsection.

AURORA's attack signatures consist of two parts. The first part is used to detect the attack payloads of stack smashing BOAs. The second part is used to detect return-into-libc BOAs. The first part is almost the same as a normal signature except the length it adopts to define a **NOP** sled. The length used in attack signatures to define a **NOP** sled is much less than the length used in normal signatures. This makes it more difficult for attackers to bypass attack signatures.

If a twelve-byte long substring of an input string satisfies the following conditions, then AURORA's attack signatures will deem it as an attack payload of a return-into-libc.

- The twelve-byte long substring could be further divided into three four-byte long substrings.
- The first four-byte long substring could be interpreted as an address in the code segment or library segment of an attack process.
- The third four-byte long substring could be interpreted as an address in the stack segment.
- The second condition is used to catch the address of the function that carries out an attack. The third condition is used to catch the first parameter of the above function. Usually the first parameter is the address of a program name string which is often stored at the stack segment. Hence, AURORA utilizes the same criteria used to find the deviation address of a stack smashing BOA to infer the address of the program name string.

Based on the above signatures, taxonomy of hosts, and push-effect, AURORA utilizes the algorithm described in Fig. 1 to implement its protection mechanism.

3.4 Criteria Used to Filter out NOP Sleds and Deviation Addresses

AURORA's signatures utilize **NOP** sleds and deviation addresses to detect BOA

```
AURORA()
{start:
   while(true)
   {use push-effect to implement a light-weight ASLR on the new process;
      while(true)
      {fork a new process/thread to serve next service request;
        receive a service request from a remote machine client_host;
        switch(client_host.type)
        {case normal host:
            if (normal signatures detect attack payload || the process crashes)
            {++client_host.counter;
               if(client_host.counter > attack_threshold)
                 client_host.type = attack host;
               close the related socket and connection and goto start;
            }
            break;
          case attack host:
            if (attack signatures detect attack payload || the process crashes)
            {++client_host.counter;
               if (client_host.counter > crash_threshold)
                   client_host.type = crash host;
               close the related socket and connection and goto start;
            }
            break;
          case crash host:
            close the related socket and connection and goto start;
        }
      }
   }
}
```

Fig. 1. AURORA algorithm.

strings. Therefore, their ability to successfully filter out the **NOP** sled and the deviation address from an attack string decides its power. A **NOP** sled has two major parts, its constituent components and its length (*i.e.* the repeated times of the constituent components in the **NOP** sled). AURORA decides the length of a **NOP** sled based on the results of a sequence of tests. The minimum length with both low false positives and low false negatives is the suggested length to the system administrators. But they can tune the length based on their requirements. The constituent components of a **NOP** sled are all one-byte non-privileged instructions. The value of the deviation address of a BOA string is not fixed due to push-effect. Hence, AURORA develops a novel approach to zero the deviation address inside a BOA attack payload.

Our experience and experiments show that when a program receives a string (including a BOA string) from outside, the program usually will process it quickly. Hence, the execution time of the function whose stack frame contains the overflowable buffer of a BOA usually is close to the execution time of the function which executes the `read` or `recv` system call. In addition, according to [60, 61] the average stack frame size of a C function is 28 bytes. Thus, if an input buffer is located at the stack segment, then it is usually close to the overflowable buffer. And if the input buffer is located at a non-stack segment, then the overflowable buffer is usually close to the caller `ebp` field of the stack frame of the function executing the `read` or `recv` system call. Moreover, the attack payload of a control-hijacking BOA is usually stored at an overflowable buffer. Due to the above phenomenon, AURORA uses the input buffer address or the caller `ebp` field address of the stack frame of the function executing the `read` or `recv` system call to infer the address of the related overflowable buffer.

Based on the input buffer address of an input string or the caller `ebp` field address of the stack frame of the function executing the `read` or `recv` system call, AURORA creates an address range, called *Suspected Deviation Address Range* (SDAR), for the input string. If a substring of the input string could be interpreted as an address falling into the SDAR of the input string, it is deemed as a deviation address. Thus, under AURORA, each input string has its SDAR whose upper boundary is formed by adding an upward offset to the *base reference point* of the input string and whose lower boundary is formed by deducting a downward offset to the same base reference point. If the input buffer of an input string is a stack buffer, the base reference point of the string is the input buffer address. If the input buffer of an input string is a non-stack buffer, the base reference point of the string is the caller `ebp` field address of the stack frame of the function executing the `read` or `recv` system call. Figs. 2 and 3 show the SDAR of the former and the latter respectively.

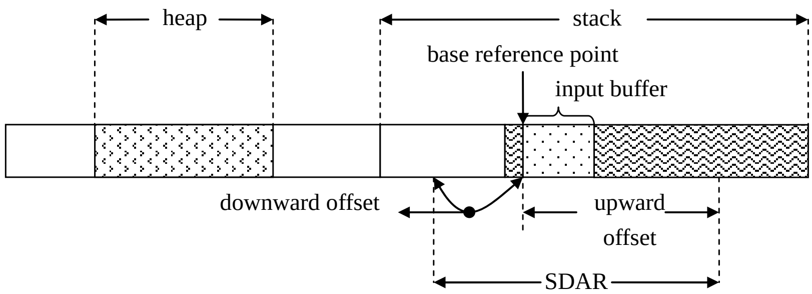

Fig. 2. The SDAB of an input string whose input buffer is a stack buffer.

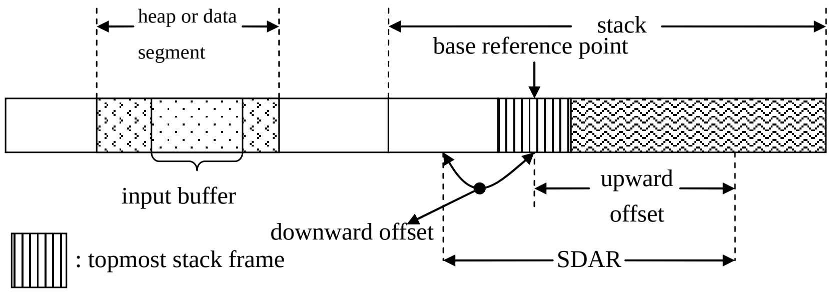

Fig. 3. The SDAB of an input string whose input buffer is a non-stack buffer.

We further analyzed the source code [29, 30, 31, 32, 33, 34, 35, 36, 37] of 9 programs [20, 21, 22, 23, 24, 25, 26, 27, 28] that have buffer overflow vulnerabilities to calculate the upward and downward offsets. Table 2 shows the programs, input buffer

Table 2. Programs with control-hijacking buffer overflow vulnerabilities, their input buffer types, distances, and overflowable buffer sizes.

	Program Name	CVE	Input buffer type	Distance (bytes)	Overflowable buffer size
1	cfengine-2.0.7	CVE-2004-1701	stack buffer	0	4096
2	pptpd-1.0.1	CAN-2003-0213	stack buffer	0	220
3	MPlayer-1.0pre5	CVE-2004-1309	stack buffer	0	102400
4	micq-0.4.0	NA	stack buffer	1202	1024
5	monkey-0.6.1	NA	stack buffer	10304	10240
6	gopher-3.0.5	CVE-2003-0805	stack buffer	– 289	256
7	thttpd-2.21	CVE-2003-0899	non-stack buffer	1192	1000
8	sniffit.0.3.7.beta	CVE-2000-0343	non-stack buffer	35848	250
9	snort-2.4.2	CVE-2005-3252	non-stack buffer	2328	1024

types, distances, and overflowable buffer sizes of these programs. In this table, if the input buffer of a vulnerable program is a stack buffer, then the "distance" column of that program represents the difference between the address of the related overflowable buffer and the input buffer address. A positive distance plus the size of the overflowable buffer represents an upward offset. The absolute value of a negative distance represents a downward offset. A zero distance means that the input buffer is also the overflowable buffer. If the input buffer of a vulnerable program is a non-stack buffer, then its distance column is calculated in a similar way, but the input buffer address is replaced by the caller `ebp` field address of the stack frame of the function executing the `read` or `recv` system call.

Analyses of Table 2 show that when the input buffer of an input string is a stack buffer, the maximum upward offset is 20544 (10304 + 10240) bytes and the downward offset is 289 bytes. Further analyses show that when the input buffer of an input string is a non-stack buffer, the maximum upward offset is 36098 (35848 + 250) bytes and the downward offset is 0 byte. In the evaluation section, we make our experiments based on these data.

4. EVALUATION

In order to evaluate AURORA's effectiveness and efficiency, various tests were designed and made. The computer used in these tests had a 3.00 GHz Pentium 4 CPU. The RAM size was 512MB. The bandwidth of the test network was 10Mb. The operating system kernel version was Linux 2.6.18.

4.1 Effectiveness Evaluation

Effectiveness was evaluated by measuring the false positives and false negatives created by AURORA. For the false negative tests, AURORA can detect all attacks targeting at programs listed at Table 2 because AURORA derived its parameters from these programs. In order to show the effectiveness of AURORA, in the false negative tests, we further collected a completely different and newer set of vulnerable programs (see Table 3) to make our experiments. The result shows that AURORA could accurately block all BOAs and allow the attacked programs to terminate normally. Besides, we also obtained

Table 3. Programs with control-hijacking buffer overflow vulnerabilities, their input buffer types, and SDAR sizes.

	program	CVE	input buffer type	SDAR size
1	3proxy 0.5.3g	CVE-2007-2031	non-stack buffer	2059
2	webdesproxy 0.0.1	CVE-2007-2668	non-stack buffer	280
3	samba-3.0.21	CVE-2007-2446	non-stack buffer	1072
4	corehttp 0.5.3alpha	NA	non-stack buffer	552
5	Elm 2.5 PL7	CVE-2005-2665	stack buffer	4792
6	fenice-1.10	CVE-2006-2022	stack buffer	1056
7	Borland InterBase LI-V8.0.0.53	CVE-2007-5243 CVE-2007-5244	stack buffer	704

20 pieces of encrypted shellcode generated by Metaspolit framework [75]. And then we combined these pieces of shellcode with 20 NOP-equivalent instructions (such as, 0×90, 0×41, and other single-byte non-privilege instructions) and a deviation address to create 60 attack strings. And we used these 60 attack strings to attack a vulnerable service. Experimental results show AURORA can detect all these attack strings correctly.

For the false positive tests, because the number of possible input types of most application programs is limited, test results from these applications are not representative. Hence, we used a `ftp` program whose input can have any type of forms to test the **worst** case false positive of AURORA.

We used `pure-ftpd` to receive randomly chosen files with total size of 13.8 GB to check AURORA's false positives. The test files consisted of 280 MB `exe` files (790), 1.18 GB `avi` and `rmvb` files (2), 409 MB `pdf` files (210), 2.08 GB `mp3` files (752), 3.78 GB `jpg` and `png` files (15471), 4.18 GB `rar` and `bz2` files (3), and 21.7 MB `dll` files (81). The number in a parenthesis represents the number of files of the corresponding file type. Totally there were 17582 files.

`pure-ftpd` uses a stack buffer to accept input files; hence, the input buffer of this program is a stack buffer. According to the rules discussed in section 3.4, the SDAR of the input strings of this program has 20544 as its upward offset and 289 as its downward offset. Based on the above SDAR, we made our tests using different **NOP** sled lengths. Fig. 4 displays the results. This figure shows the numbers of false positive files and the related **NOP** sled lengths. For example, when AURORA used 60 as its threshold to define a **NOP** sled, the number of files that AURORA misjudged as containing an attack payload was 9 which is only 0.05% of the 17582 test files.

As Fig. 4 shows, when the length used to define a **NOP** sled decreases, the number of false positive files increases. The appropriate length used in normal signatures to define a **NOP** sled is the one that has both low value and low false positives so that normal traffic will not frequently erroneously be blocked and attackers could not bypass the detection of AURORA easily. Since attack signatures are applied to hostile hosts, the length used to define a **NOP** sled is supposed to be small so that it will be quite difficult to be bypassed. But a higher false positive rate is endurable. Based on the above principle, system administrators could choose the **NOP** sled length that is most suitable for their systems. For example, a system administrator could use 60 as the length criterion to define a **NOP** sled in the normal signatures and 20 as the length criterion to define a **NOP** sled in the attack signatures. Even though length 60 does not look like a small number, comparing with the

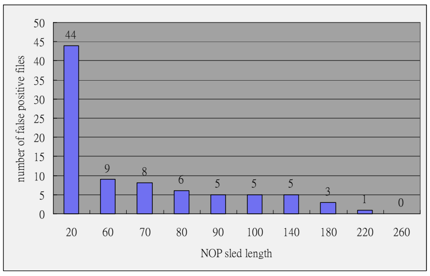

Fig. 4. Number of false positive files with different **NOP** sled lengths.

NOP sled lengths listed in Table 1, it is a small value. In addition, AURORA uses push-effect to implement a light-weight ASLR. This approach can force attackers to use a long **NOP** sled in their attack payloads, because short **NOP** sleds are very likely to crash attacked processes.

4.2 Performance Analysis

To test the micro-benchmark of AURORA, we wrote a sender and receiver program on two different machines and measured the time it took to transmit data. The two test machines were connected through a 10Mb line directly. The sender and receiver communicated with each other through a TCP connection. The sender continued pumping data into the receiver. The receiver read input characters using buffers with size 4KB, 8KB, 16KB, and 32KB. Then we made an experiment to calculate the time that a tested system took to read the input characters 1,000,000 times. The above experiment was repeated 10 times. We used the average result of these 10 experiments to represent the time that a tested system took to execute a `read` system call 1,000,000 times. Table 4 shows the results.

To test the macro-benchmark of AURORA, we used a `ftp` program to retrieve a 35.4 MB `pdf` file 100 times and measured the total time spent in transmitting these files and the overheads. Table 5 shows the results. The overhead is only about 1%.

Table 4. Micro-benchmark results with various input buffer sizes. The "avg. time" row represents the average time a tested system took to execute the `read` system call 1,000,000 times.

Buffer size	4KB		8KB		16KB		32KB	
	Original	AURORA	Original	AURORA	Original	AURORA	Original	AURORA
Avg. time (sec)	186.47	242.32	334.59	439.08	686.86	844.54	1281.011	1712.26
Overhead	0.300		0.312		0.230		0.337	

Table 5. Macro-benchmark results. The "total time" row represents the total time spent to retrieve a 35.4MB pdf file 100 times.

	Normal host	AURORA
Total time (sec)	404	408
Overhead	0.0099	

4.3 Attack Analysis and Future Work

Input strings stored in successive memory areas are analyzed by AURORA as a single string. Hence, an attacker could not bypass AURORA's detection by splitting her/his attack string into several small substrings. Therefore, AURORA could defend a system against various control-hijacking BOAs correctly. But current AURORA version is not designed to detect a BOA whose overflowable buffer is not located at the stack segment of a process. This type of attacks usually takes over a host through overwriting a global function pointer stored at the data segment of a process. Solving this problem will be our future work.

5. CONCLUSION

In this paper, we propose a solution, AURORA, to the notorious security problem – control-hijacking buffer overflow attacks. AURORA developed an accurate adaptive detection mechanism that can recognize and block stack-smashing BOA strings and return-into-libc attack strings, including zero-day ones, inside the kernel. Hence, it can prevent them from damaging the user mode stack of the attacked process. AURORA is an OS-based solution; thus, it does not need to modify the source code of any application programs. In addition, AURORA is a signature generation-free solution. Therefore, it doesn't need to maintain any new signatures, once it has been installed. AURORA allows an attacked process to continue its execution without the side effect of process idleness or repeated process crashes. Furthermore, AURORA is compatible with existing operating systems and application programs. Finally, AURORA could work with various protection mechanisms to provide an extra layer of protection. Experiment results showed that with less than 1% overhead and negligible false positives, AURORA accurately blocked stack-smashing BOAs and return-into-libc attacks.

REFERENCES

1. CERT, http://www.us-cert.gov/.
2. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer overflow attacks," in *Proceedings of USENIX Security Symposium*, 1998, pp. 63-78.
3. Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *Proceedings of Annual Computer Security Applications Conference*, 2006, pp. 429-438.
4. Bulba and Kil3r, "Bypassing StackGuard and StackShield," *Phrack Magazine*, Vol. 10, 2000, http://www.phrack.org/issues.html?issue=56&id=5#article.
5. F. Hsu, F. Guo, and T. Chiueh, "Scalable network-based buffer overflow attack detection," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2006, pp. 163-172.
6. Ethereal: A Network Protocol Analyzer, http://www.ethereal.com.
7. S. Bhatkar, D. DuVarney, and R. Sekar, "Address obfuscation: An efficient ap-

proach to combat a broad range of memory error exploits," in *Proceedings of USENIX Security Symposium*, 2003, pp. 105-120.
8. The PaX Address Space Layout Randomization Project, http://pax.grsecurity.net/.
9. Aleph One, "Smashing the stack for fun and profit," *Phrack Magazine*, Vol. 7, 1996, http://www.phrack.org/issues.html?issue=49&id=14.
10. G. Kc, A. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003, pp. 272-280.
11. StackShield, http://www.angelfire.com/sk/stackshield.
12. G. Richarte, "Four different tricks to bypass StackShield and StackGuard protection," Core Security Technologies, 2002, http://www.coresecurity.com/files/attachments/StackGuard.pdf.
13. N. Sovarel, D. Evans, and N. Paul, "Where's the FEEB? The effectiveness of instruction set randomization," in *Proceedings of USENIX Security Symposium*, 2005, pp. 145-160.
14. SolarDesigner, "Non-executable user stack," http://www.openwall.com/linux.
15. R. Wojtczuk, "Defeating solar designer's non-executable stack patch," http://insecure.org/sploits/non-executable.stack.problems.html.
16. C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "Pointguard: Protecting pointers from buffer overflow vulnerabilities," in *Proceedings of USENIX Security Symposium*, 2003, pp. 91-104.
17. ProPolice, http://www.x.org/wiki/ProPolice.
18. T. Chiueh and F. Hsu, "RAD: A compiler time solution to buffer overflow attacks," in *Proceedings of International Conference on Distributed Computing Systems*, 2001, pp. 409-419.
19. S. Bellovin, "Distributed denial of service attacks," http://www.research.att.com/smb.
20. MARC, http://marc.info/.
21. Full-Disclosure, https://lists.grok.org.uk/mailman/listinfo/full-disclosure.
22. National Vulnerability Database, http://nvd.nist.gov/.
23. Derkeiler, http://www.derkeiler.com/.
24. SECUREROOT, http://www.secureroot.com/.
25. Chinese IT Headquarters, http://www.ie100.cn/.
26. iDefense Labs, http://labs.idefense.com/.
27. Security tracker, http://www.securitytracker.com/.
28. SECWATCH.ORG, http://www.secwatch.org/.
29. Tengu.be, http://www.tengu.be/index.php.
30. Thttpd, http://www.acme.com/software/thttpd/.
31. Cfengine, http://www.cfengine.org/download.phtml.
32. LScube, http://live.polito.it/.
33. Gopher, http://gopher.quux.org:70/devel/gopher/Downloads/old.
34. Micq, http://linux.maruhn.com/sec/micq.html.
35. Monkeyd, http://monkeyd.sourceforge.net/.
36. Mplayer, http://www1.mplayerhq.hu/MPlayer/releases/MPlayer-1.0pre5.tar.bz2.
37. Pptpd, http://www.poptop.org/.
38. F. Hsu and T. Chiueh, "CTCP: A transparent centralized TCP/IP architecture for network security," in *Proceedings of the 20th Annual Conference on Computer Se-*

curity Application, 2004, pp. 335-344.
39. J. Pincus and B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," in *Proceedings of IEEE Symposium on Security and Privacy*, 2004, pp. 20-27.
40. D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proceedings of International Conference on Software Engineering*, 2006, pp. 162-171.
41. GCC extensions, http://gcc.gnu.org/extensions.html.
42. Exec Shield, http://en.wikipedia.org/wiki/Exec_Shield.
43. C. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. Debray, and J. Hartman, "Protect against unexpected system calls," in *Proceedings of USENIX Security Symposium*, 2005, pp. 239-254.
44. Nergal, "The advanced return-into-lib(c) exploits: PaX case study," *Phrack Magazine*, Vol. 11, 2001, http://www.phrack.org/issues.html?issue=58&id=4.
45. T. Toth and C. Kruegel, "Accurate buffer overflow detection via abstract payload execution," in *Proceedings of International Symposium on Recent Advances in Intrusion Detection*, 2002, pp. 274-291.
46. D. Ruiu, "NOP equivalent opcodes for shellcodes – Canonical List," http://www.cansecwest.com/noplist-v1-1.txt.
47. M. Dalton, H. Kannan, and C. Kozyrakis, "Real-world buffer overflow protection for userspace and kernelspace," in *Proceedings of USENIX Security Symposium*, 2008, pp. 395-410.
48. J. Xu, Z. Kalbarczyk, and R. Iyer, "Transparent runtime randomization for security," in *Proceedings of Symposium on Reliable Distributed Systems*, 2003, pp. 260-269.
49. C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006, pp. 339-348.
50. E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanovic, "Randomized Instruction Set Emulation," *ACM Transactions on Information and System Security*, Vol. 8, 2005, pp. 3-40.
51. O. Ruwase and M. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of Network and Distributed System Security Symposium*, 2004, pp. 159-169.
52. W. Cui, M. Peinado, H. Wang, and M. Locasto, "ShieldGen: Automatic data patch generation for unknown vulnerabilities with informed probing," in *Proceedings of IEEE Symposium on Security and Privacy*, 2007, pp. 252-266.
53. D. Wagner, J. Foster, E. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proceedings of Network and Distributed System Security Symposium*, 2000, pp. 3-17.
54. B. Chess, "Improving computer security using extended static checking," in *Proceedings of IEEE Symposium on Security and Privacy*, 2002, pp. 160-173.
55. D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *Proceedings of USENIX Security Symposium*, 2001, pp. 177-190.
56. R. Rugina and M. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," *ACM Transactions on Programming Languages and Systems*, Vol. 27, 2005, pp. 185-235.

57. S. Bhatkar, R. Sekar, and D. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of USENIX Security Symposium*, 2005, pp. 255-270.
58. A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. Li, R. Kuo, and K. Fan, "Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities," in *Proceedings of IEEE/IFIP Network Operations and Management Symposium*, 2004, pp. 235-248.
59. X. Wang, C. Pan, P. Liu, and S. Zhu, "SigFree: A signature-free buffer overflow attack blocker," in *Proceedings of USENIX Security Symposium*, 2006, pp. 225-240.
60. D. Ditzel and R. McLellan, "Register allocation for free: The C machine stack cache," in *Proceedings of Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982, pp. 48-56.
61. S. Cho, P. Yew, and G. Lee, "Decoupling local variable accesses in a wide-issue superscalar processor," in *Proceedings of Annual International Symposium on Computer Architecture*, 1999, pp. 100-110.
62. L. Li, J. Just, and R. Sekar, "Address-space randomization for windows systems," in *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006, pp. 329-338.
63. S. Singh, C. Estan, G. Varghese, and S. Savage, "Automated worm fingerprinting," in *Proceedings of Symposium on Operating Systems Design and Implementation*, 2004, pp. 45-60.
64. H. Kim and B. Karp, "Autograph: Toward automated, distributed worm signature detection," in *Proceedings of USENIX Security Symposium*, 2004, pp. 271-286.
65. C. Kreibich and J. Crowcroft, "Honeycomb − Creating intrusion detection signatures using honeypots," *ACM SIGCOMM Computer Communication Review*, Vol. 34, 2004, pp. 51-56.
66. J. Newsome, B. Karp, and D. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *Proceedings of IEEE Symposium on Security and Privacy*, 2005, pp. 226-241.
67. V. Yegneswaran, J. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," in *Proceedings of USENIX Security Symposium*, 2005, pp. 97-112.
68. M. Costa, J. Crowcroft, M. Castro, and A. Rowstron, "Vigilante: End-to-end containment of internet worms," in *Proceedings of the Symposium on Operating Systems Principles*, 2005, pp. 133-147.
69. J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Network and Distributed System Security Symposium*, 2005, pp. 174-198.
70. Z. Liang and R. Sekar, "Fast and automated generation of attack signatures: A basis for building self-protecting servers," in *Proceedings of ACM Conference on Computer and Communications Security*, 2005, pp. 213-222.
71. R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif, "Misleading worm signature generators using deliberate noise injection," in *Proceedings of IEEE Symposium on Security and Privacy*, 2006, pp. 17-31.
72. M. Polychronakis, K. Anagnostakis, and E. Markatos, "Network-level polymorphic shellcode detection using emulation," in *Proceedings of the 3rd Conference on De-*

tection of Intrusions and Malware and Vulnerability Assessment, 2006, pp. 54-73.
73. P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis, "STRIDE: Polymorphic sled detection through instruction sequence analysis," in *Proceedings of IFIP International Conference on Information Security*, 2005, pp. 375-392.
74. Solar Designer, "Getting around non-executable stack (and fix)," *Bugtraq Mailing List*, http://seclists.org/bugtraq/1997/Aug/63.
75. Metaspolit, "Penetration testing resources," http://www.metasploit.com/.

Li-Han Chen (陳立函) is a Ph.D. student in the Department of Computer Science and Information Engineering of National Central University. He received his M.S. degree in Computer Science and Information Engineering from National Central University, Taoyuan, Taiwan, in 2008, and his B.S. degree in Chemical Engineering from National Tsing Hua University. His research areas include mobile security, operating system, and network security.

Fu-Hau Hsu (許富皓) is an Assistance Professor at National Central University and have had an appointment in the Department of Computer Science and Information Engineering since August 2005. He is affiliated with the Advanced Defense Lab and the Wireless and Multimedia Lab.

Cheng-Hsien Huang (黃政憲) received the B.S. degree in Computer Science from Chung Cheng Institute of Technology, Taoyuan, Taiwan, in 1996, and the M.S. degree in Electrical Engineering also from CCIT in 2001. He is currently a Ph.D. student in the Department of Computer Science and Information Engineering of National Central University. His current research interests include attack prevention from Linux kernel and malware classification. His past work experience include network management and application development.

Chih-Wen Ou (歐智文) was the Ph.D. student of Computer Science and Information Engineering in National Central University from September 2008 to June 2009, and received his B.S. degree in Computer Science in National Chiao Tung University in June 2006, and his M.S. degree in Computer Science and Information Engineering in National Central University in July 2008. His researches are security issues about OS design, network-based threat and mobile devices, especially windows mobile and android. He is an assistant researcher now in ICS Lab of Telecommunication Laboratories of Chunghwa Telecom Co. and responsible for developing NFC related libraries on Windows Mobile, Android and iPhone since September 2009.

Chia-Jun Lin (林佳潤) graduated with a Master degree of Computer Science and Information Engineering from Advanced Defense Lab in National Central University, Taiwan. His main fields of research are security, including Linux and Windows kernel, reverse engineering, web security, and mobile security.

Szu-Chi Liu (劉思奇) received his M.S. degree in Computer Science and Information Engineering from National Central University, Taoyuan, Taiwan, in 2009, and his B.S. degree in Mathematics from National Tsing Hua University. His research interests include Linux kernel programming and reputation system.