Sheffield Hallam University

Protecting Against Address Space Layout Randomization (ASLR) Compromises and Return-to-Libc Attacks Using Network Intrusion Detection Systems

DAY, David and ZHAO, Zhengxu

Available from Sheffield Hallam University Research Archive (SHURA) at:

http://shura.shu.ac.uk/5233/

This document is the author deposited version. You are advised to consult the publisher's version if you wish to cite from it.

Published version

DAY, David and ZHAO, Zhengxu (2011). Protecting Against Address Space Layout Randomization (ASLR) Compromises and Return-to-Libc Attacks Using Network Intrusion Detection Systems. International Journal of Automation and Computing, 8 (4), 472-483.

Repository use policy

Copyright © and Moral Rights for the papers on this site are retained by the individual authors and/or other copyright owners. Users may download and/or print one copy of any article(s) in SHURA to facilitate their private study or for non-commercial research. You may not engage in further distribution of the material or use it for any profit-making activities or any commercial gain.

Sheffield Hallam University Research Archive
http://shura.shu.ac.uk

Protecting Against Address Space Layout Randomization (ASLR) Compromises and Return-to-Libc Attacks Using Network Intrusion Detection Systems

David J Day
School of Computing and Mathematics
University of Derby
Derby, UK.
d.day@derby.ac.uk

Zhengxu Zhao[1]
Faculty of Information Science and Technology
Shijiazhuang Tiedao University
Shijiazhuang Hebei, China.
zhaozx@sjzri.edu.cn

Abstract: Writable XOR eXecutable (W ⊕ X) and Address Space Layout Randomisation (ASLR), have elevated the understanding necessary to perpetrate buffer overflow exploits [1]. However, they have not proved to be a panacea [1] [2] [3] and so other mechanisms such as stack guards and prelinking have been introduced. In this paper we show that host based protection still does not offer a complete solution. To demonstrate, we perform an over the network brute force return-to-libc attack against a pre-forking concurrent server to gain remote access to a shell. The attack defeats host protection including W ⊕ X and ASLR. We then demonstrate that deploying a NIDS with appropriate signatures can detect this attack efficiently.

Keywords : Buffer overflow, Stack overflow, IDS, Signature, Rules, Return-to-libc, Attack, Pre-forking

I. INTRODUCTION

Nearly all internet worms are facilitated through the exploit of buffer overflow vulnerabilities [4] and the threat of buffer overflow exploits continues to dominate as the most severe and frequent [5]. Buffer overflow vulnerabilities have been exploited for over 20 years and continue to evolve [6] despite innovative progress with host based protection mechanisms.

Buffer overflow attacks are made possible through absent or erroneous bounds checking of user input data. These vulnerabilities only exist when developing software using languages which do not enforce run-time bounds checking such as C and C++. These two languages account for more software than any other [7], exacerbating the problem.

The software industry has responded to these types of attacks by releasing patches for their applications. These code corrections are released at a point where the vulnerability becomes known, usually after it has been penetrated, and this leads to a patch-penetrate cycle of software security [8]. Unfortunately this treats the symptom rather than the underlying cause. Consequently systems remain vulnerable to attacks perpetrated prior to the software vendor being aware of any vulnerability (zero day attacks).

A. *Buffer overflow protection*

Since the first reported buffer overflow attack, the Morris worm in 1988 [9], system designers have been developing protection mechanisms to eradicate them. Most have proposed host based protection mechanisms which prevent changes to program execution flow e.g. StackGuard [10] and Propolice [11]. Other techniques involve modifying the CPU and operating system e.g. ASLR and W⊕X. However, while the safeguards have raised the bar significantly, the attackers continue finding creative ways to defeat them. Reactive protection mechanisms cannot prevent human error, thus the solution may be better design and testing of software or the use of languages that enforce run-time bounds checking. This philosophy is creditable but also expensive [12], and is unlikely to be done at the cost of performance [13]. It seems almost inevitable that buffer overflows will continue to emerge as a result of human error either via the generation of new vulnerable code or the re-use of legacy vulnerable code. In either case the root cause is putting performance before security.

B. *NIDS and Shell code detection*

A popular method of mitigating the risk of buffer overflow attacks is through the use of Network Intrusion Detection Systems (NIDS). NIDS's monitor network traffic for suspicious activity by examining packets for patterns indicative of known exploits. This is performed by placing the systems at key points within the network, to scan as much relevant inbound and outbound traffic as necessary [14]. Many inventive proposals have be made in this area including the use of Artificial Intelligence to predict attacks [15]

Often malicious parties intend to gain remote access to a system via a system shell. They can then perform a number of malicious activities, including the introduction of root kits, which facilitate easier future access to the remote system. As a result various IDS rules have been developed to detect the injection of shell code into applications [16] [17]. Most of these involve techniques to either detect or obfuscate shell code, respectively.

[1]Corresponding Author: Zhengxu Zhao

1) Present NIDS and Shell code detection limitations

The release dates of rules for software vulnerabilities are often close to that of the patch. Hence the application may be susceptible to zero day attacks [18].

In addition, other attack methods which make use of buffer overflow vulnerabilities also exist e.g. return-to-libc. These attacks use code already loaded in memory and do not need to inject shell code [5]. This renders many shell code detection rules useless. Wide scale host based protection mechanisms such as (W ⊕ X) and ASLR have been implemented to prevent these types of attacks, however scenarios exist where they can be perpetrated. One of these, brute force attacks against pre-forking daemons [2], will be the primary focus of this paper

C. Organisation of paper

The remainder of this paper is constructed as follows. Section II discusses the threat posed by buffer overflow attacks and how mainstream protection mechanisms attempt to protect against them. In Section III, we explain and demonstrate through simulation, how the implementation of (W⊕X) and ASLR still leaves a residual threat. This threat, namely that posed by pre-forking concurrent servers, is proved through a demonstration of our own brute force return-to-libc attack. In Section IV we exhibit protection mechanisms that attempt to prevent this type of attack variant, discussing their efficiency and short comings. In Section V we explain the role Intrusion Detection Systems might play in obviating these attacks and create some generic rules, implemented in Snort [19], which could be used to detect them. We then test these rules using real-world traffic and report our findings in section VI. In section VII, we offer our conclusions.

II. BUFFER OVERFLOW ATTACKS AND PROTECION MECHANISMS

Structural programming languages such as C use procedures and functions to alter the flow of execution of a program, this takes place when a procedure is "called". Procedures and functions may have their own local variables, allocated at runtime, along with other variable values that are passed into their parameters as arguments. They may also return values to the calling procedure when necessary. When a procedure or function has finished processing, the path of execution will return to the point immediately after the instruction which called the procedure. The stack is used to keep track of the flow of program execution and the procedure's local variables and parameters. In essence it is used as temporary storage with values pushed onto it when the function/procedure is called (the prologue) and popped off when the procedure returns (the epilogue) [20]. The information stored on the stack for a called procedure is referred to as its stack frame. The following figure shows the stack layout after a code injection buffer overflow attack has taken place. The extensible base pointer (EBP) or frame pointer, is used to locate components on the stack frame as offsets, and NOPS are assembler operations that do not perform any operation other than moving to the next instruction in the sequence.

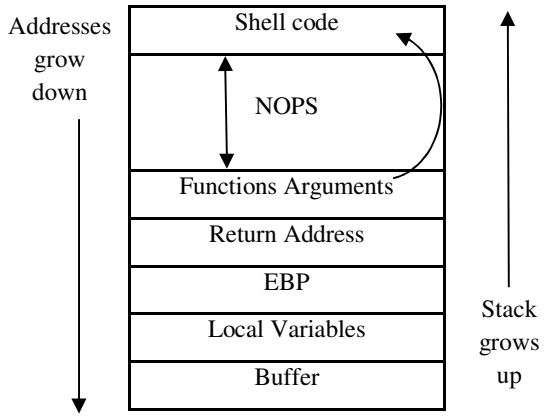


Fig.1 Stack state after a traditional attack

In Fig. 1 memory reserved for the buffer has been flooded such that shellcode and a number of nops (no-op sled) have been injected onto the stack. In addition the return address has been overwritten with the predicted address for the shellcode. If this predicted address is inaccurate then flow of execution may be resumed from an area of the no-op sled where it will proceed through each no-op instruction until it ultimately reaches and then executes the shellcode. Thus the flow of execution would be re-directed as a result of the vulnerable functions epilog.

The same result can be achieved by modifying function pointer arguments which point to the address of a function [21] or by changing the saved frame pointer to point to a frame with a compromised return address [22]. These vulnerabilities are available since C and C++ [23] allow the use of arrays and pointers without bounds checking, when this is combined with the c-libraries dangerous string functions e.g. strcpy, strcat, sprint, gets, which terminate based on a null character rather than a defined number of bytes, the buffer can be overflowed.

A. *Host based protection mechanisms*

Host based protection consist of compilation, CPU, and operating system mechanisms. The most prominent are W ⊕ X, ASLR and stack based buffer overrun. These protection mechanisms along with the attacks that are designed to obviate their functionality are discussed in the sections immediately following

1. W⊕X

W ⊕ X allows the processor to mark memory locations which should not contain executable code, e.g. the stack and heap, as Write XOR eXecute [24]. That is they can be written to or executed, but not both. It is also referred to as Data Execution Prevention (DEP). The intention is to ensure that the return address would only

point to the address of trusted code [25]. However it can be bypassed using the return-to-libc method.

a) Return-to-libc attack

This attack uses addresses of c-library functions already loaded into memory. Thus it avoids placing executable code on the stack, evading W ⊕ X. An example of the state of the stack is shown in fig. 2.

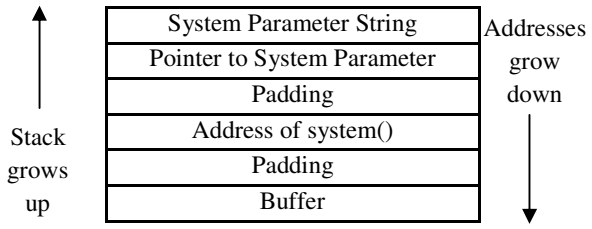


Fig 2. Stack of stack after a return-to-libc attack

Fig. 2 shows that once the vulnerable function returns it will execute the system function and parameter.

Defeating W⊕X using the return-to-libc technique inspired the creation of ASLR [26].

2) Address Space Layout Randomisation

ASLR randomises the base address of the stack, heap, code, memory mapped segments of executables, and dynamic libraries at load and link time [27]. Return-to-libc attacks are defeated through c-library address randomisation at execution. However, this isn't entirely dependable, elements of the operating system may be protected, but not all third party applications are appropriately compiled for ASLR and remain vulnerable to existing code attacks such as dynamic link library (dll) and binary trampolining [28].

a) dll and binary trampolining

Due to oversight, incompatibilities, or in an effort to increase performance, some applications aren't compiled to use ASLR. The binary processes address space, or related dll's, will contain known addresses of operator codes. If they contain a buffer overflow vulnerability, these addresses can be injected onto the stack and used to change the flow of program execution. These types of attack can be foiled by W⊕X or stack based buffer overrun protection.

3) C Library Address Posistioning

The Openwall Project [29] produced a Linux kernel patch which ensures the c-library address is loaded into memory under 0x10000000 [30]. This affords some protection from return-to-libc attacks as the c-library function address to be injected will contain null bytes. This can cause malicious strings to terminate prematurely. The patch was released in September 2002 [29] yet it has not implemented by default in most Linux distributions e.g. Ubuntu 9.

4) Stack based buffer overrun protection

Stack based buffer overrun protection adds a compilation stage transforming the program in an attempt to meet the ideal stack model [22], see fig. 3. An interpretation of this, Stack Smashing Protection (SSP), has been included in GCC since version 4.1 [31].

In fig. 3, the stack is shown amended compared to fig. 1. A guard is placed on the stack prior to the buffer to protect the values preceding it from an overflow i.e. the frame pointer, return address and function's arguments. This is facilitated by GCC recording the size of the buffer and adding specific code to the object. This code inserts a guard and guard inspection mechanism. Alteration of the guard at runtime causes the process to terminate with an error message. In addition, local variables are arranged on the stack after the buffer, thus protecting function pointers from overwrites [22].

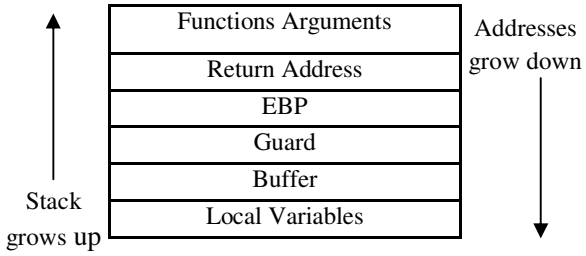


Fig.3 Ideal stack model

This defense is not flawless however. A common method of circumnavigating this is via SEH (Structured Exception Handling) attacks [32] .

III. PREFORKING SERVERS OFFERING A RESIDUAL BUFFER OVERFLOW THREAT

ASLR can leave an attacker with little choice but to guess the addresses of commands needed to perpetrate a return-to-libc attack. However, if an attacker launches an attack against a vulnerable pre-forking server, such as that used by the Oracle 9 PL/SQL Apache module [33], they can utilize the fact that spawned child processes inherit the same virtual address space as that of their parent process. As such they are suitable to brute force return-to-libc attacks. The format of this type of attack was first laid out by Shacham [2] in 2004, and is explained here for the purpose of providing both additional detail and context.

A pre-forking concurrent server operates by pooling a number of listening processes at start up. This offers superior performance to alternative concurrent server designs e.g. handling requests iteratively, or spawning a child process for each new client request [34]. The benefits make it a very popular method of handling requests for http, imap and smtp servers.

By forking, child processes are capable of accepting new connections on the same listening socket as their parent, yet they also inherit the same virtual address space, see fig. 4. This leaves them vulnerable to brute force attacks that continually connect and overflow the buffer such that the return address on the stack is overwritten with a guess for the address of a specific c-library function. An incorrect guess will result in a segmentation fault which causes the child process to terminate and a new child process to be spawned in its place. Thus a process of elimination can be used to find a c-library function.

ASLR unpredictability is not extensive, on a 32 bit host this has been documented for PaX ASLR as 16bits (65536 addresses) [35]. Yet, we discovered during our experimentation with ASLR on Fedora and Ubuntu boxes that this number is significantly less. The logistics of our attack are outlined in the following sections

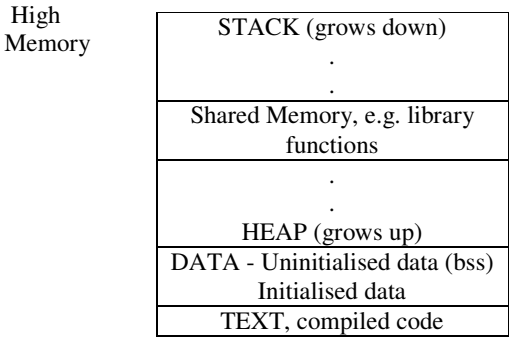


Fig.4 Virtual Address Space

A. *Test bed environment*

Our attack, inspired by Shacham [2], consists of a reconnaissance component followed by the exploit itself, the latter making use of the information gleaned from the former. The following figure shows how our test environment was assembled:

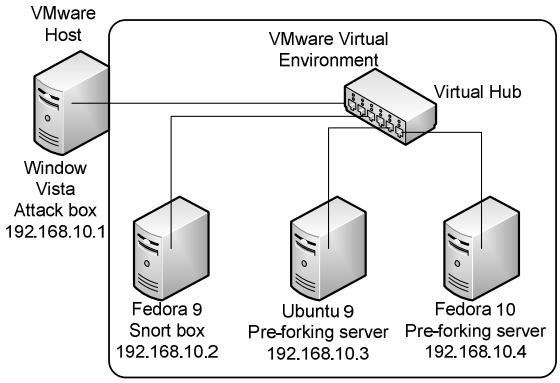


Fig. 5 Virtual Test Environment

The test environment was created using VMware workstation [36], this allowed for rapid control transfer between guest machines via tabs.

The attack box is also the host for the virtual machines and it holds the host operating system Microsoft Vista, along with the VMware software necessary to facilitate the virtual environment. VMware is configured to allow the use of a virtual hub where each of the guest machines, along with the host itself, is connected. The Vista host (attack box) has a shared folder which is configured to allow the guest machines access to custom written vulnerable pre-forking server applications. This allowed us to test Ubuntu 9 and Fedora 10 buffer overflow security mechanisms. The attack box also contains code to facilitate a brute force over the network reconnaissance and return-to-libc attack, allowing a remote shell to be opened on the victim's machine. The attack box includes Apache and the malicious application *rshell* available for download via the web server. The Snort box contains a copy of Wireshark [37] for capturing traffic to assist in rule creation, and a running copy of the IDS Snort [38] used to test and tune rules.

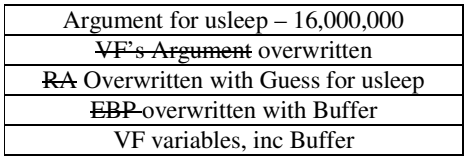
B. *Vulnerable pre-forking server (ssprocess)*

Ssprocess is a pre-forking concurrent server; it was written in C and compiled using GCC for use with a Linux operating system. The server pools a number of listening processes, this is specified by its parameter (150 child processes were selected as this is the default for the Apache web server) at start up. Each child process is capable of accepting the connection on the same listening socket and executes the vulnerable function each time a connection is accepted and data is received. This received data is copied into a buffer without performing bounds checking, and it is this simulated oversight which leaves the function open to a stack based buffer overflow attack.

C. *Reconassence application (NetClientExploit)*

NetClientExploit is the process responsible for reconnaissance. It makes a connection to the vulnerable server and overflows the buffer such that the return address is overwritten with a guessed address for the c-library function `usleep`. 8 bytes prior to this on the stack, i.e. `usleep` address guess + 8 bytes, is the parameter of the `usleep` function. The value chosen is 16,000,000 so that if the `usleep` function was guessed correctly this would cause the server system to pause for 16 seconds.

This is achieved by the attacking application looping through successive incremental guesses for the `usleep` function, repeatedly making a connection to the server and sending a crafted buffer to compromise the stack, as shown in the following figure.



VF – Vulnerable function
RA - Vulnerable function s Return Address

Fig. 6 Compromised stack

The score through sections in the figure depicts legitimate storage on the stack which has been overwritten. During a function's epilogue the following takes place

- The stack pointer is replaced with the frame pointer, also called the Extended Base Pointer (EBP) i.e. mov %ebp, %esp
- The base pointer is removed from the stack (it was placed there during the prologue) i.e. pop %ebp
- The return address is popped from the stack and program execution is redirected to this address. With a legitimate program this value would have been placed there during the "call" command for the function [39] .

Considering the vulnerable function, at this point there are two possible outcomes; if the guess is incorrect, then a segmentation fault will occur. Segmentation faults occur when a program attempts to access a disallowed memory location or attempts to access it in an inappropriate way. If a function, which is not `usleep`, is discovered then the program could attempt to execute it, however it is unlikely to execute correctly with an integer parameter of 16,000,000 and this would classify as inappropriate and a segmentation fault would occur. Once a segmentation fault occurs, *ssprocess* will terminate the child process before launching a new child process. Since the child inherits its address layout from the parent it will have an identical layout to the one previously terminated. *NetClientExploit* will respond to the closed connection by closing the socket, incrementing the guess address for `usleep` and trying again.

If the correct guess is made then execution will resume from the `usleep` function when the vulnerable function exits. During a function's prologue the following occurs [20].

- The frame pointer is pushed onto the stack i.e. pushl %ebp
- The stack pointer is copied as the frame pointer, making it the new frame pointer
- Room is made on the stack for the functions local variables.

The `usleep` function will be expecting its parameter 8 bytes up the stack relative to the EBP, if the function had been called legitimately then the parameter would have been placed there prior to the function's "call" command, however in this instance it has been maliciously injected there. As discussed `usleep` executing with the parameter 16,000,000 will cause the server to suspend activity while it waits 16 seconds, *NetClientExploit* times the amount of time taken for the vulnerable server to start accepting data on its socket and if it exceeds this then the correct address of `usleep` is deemed to have been found. The location of c-library functions always remain constant in relation to each other, thus, once the address of one function has been established it is a trivial exercise to determine the location of the others.

1) Determining ASLR entropy

ASLR is designed to randomise the base addresses of the core sections in a processes virtual address space [40]. This includes the base address of the c-library which is the target of the exploit. In order to establish the value of this offset, for any given instance of randomisation, it was necessary to manually turn off address space randomisation and then determine the base address. The former is performed using sysctl –w kernel.randomise_va_space=0 and the later established using cat /proc/<PID>/maps for a running process, where <PID> is the process identifier [41]. In the case of Linux Ubuntu 9 this is identified as /lib/libc-2.9.so and in our experiment this is at 0xb7e6f000. The offset, called delta_mmap, can be established using a debugger, breaking at `main`, and displaying the address of `usleep`, in our experiment this was 0xb7f4c110. The offset is thus 0xdd110 and will remain fixed regardless of base address randomisation.

In order to determine if the attack has been successful it is necessary to establish the theoretical maximum number of guesses needed, if this figure is surpassed the attack can be deemed to have failed. There is little to document the level of entropy employed by the various distributions of Linux. However PaX, a Linux kernel patch which facilitates ASLR, is documented as randomising the base addresses of the process address space for the executable (data and text), mapped (heap and shared libraries) and the stack as now described. A random variable is added to each area of the section in the process address space, in the case of the shared library, this variable is called delta_mmap. Since IA32 architectures use a 4k page file bits 0-11 cannot be randomised as it would interfere with the page offset. In addition the high nibble (bits 28-31) is not randomised as this is used to allow for large memory mappings. This leaves 16bits of arbitrariness for base addresses [35]. However, while PaX has been in operation since 2000 [42] it is still not enforced by default in the majority of Linux distributions, including Ubuntu 9, which is being used for the experiment. There is, however, a basic form of ASLR which has been present in many Linux distributions since RedHat enterprise version 3 [25]. This is the Arjan van de Ven's ASLR implementation. Unfortunately the level of randomness provided by this mechanism isn't implicitly publically documented, hence in order to determine this, empirical observation was performed. This was done by analyzing the output of a script which ran the unix shell command `ldd`, 2^{16} times for an arbitrary program. 2^{16} times was chosen as it is documented that Arjan Van de Ven's implementation has less randomness than PaX (16bit) therefore this should prove more than sufficient to ensure a complete output of all possible c-library locations. The output was analysed and it was observed that the c-library starts at its lowest at address 0xb7d70000 and at its highest at b7f6e000. It was also observed, as expected, that the final 12 bits are not randomised, this is due to aforementioned page offset. Hence a maximum number of 0x1FE+1 or 511 guesses is needed, as given by

$$\mathrm{m} = \left(\frac{u-l}{0x1000}\right)+1$$

Where m is the maximum number of guesses and u and l are the upper and lower limits of the start address for the c-library that can be randomly generated.

During our experiment we ensured that Linux Ubuntu 9 was running with its default security e.g. ASLR and (W ⊕ X) were in effect. We ran the *NetClientExploit* against *ssprocess* 30 times with it configured to spawn 150 child processes. The results showed a mean average of 249 guesses needed to guess `usleep`, within 2.5% of the true value of 255.5, with an average time to discover it of 13.5 seconds. Please note that the `usleep` function causes a pause for 16 seconds in this experiment hence the total average reconnaissance time was 29.5 seconds.

With the location of the `usleep` function uncovered the location of the c-library functions necessary to perform a return-to-libc attack were then calculated. Our attack makes use the function's `system` and `exit` which are 0xa38b0 and 0xaeae0 memory locations lower in the c-library than `usleep`.

D. *Perpetrating the attack*

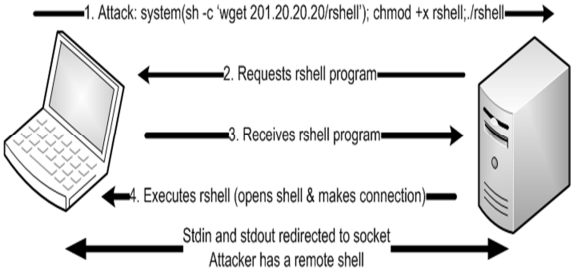


Fig. 7 Overview of malicious assault

Step1: The attack commands are pushed into the vulnerable application via a buffer and flooding the stack.
Step2: The attack string is executed causing the victim to initiate a "`wget`"
Step3: The malicious application is downloaded from the hostile system.
Step4: Once the malicious application has been downloaded and executed it sets up a connection socket to the malicious system which is listening for the incoming connection. The standard terminal input and output is redirected to the socket and a shell is opened thus passing control to the malevolent party.

1) Creating the attack string

There are two challenges which need to be overcome to create an attack string which will result in the successful perpetration of this attack. Firstly, when the vulnerable function exits, program execution must be redirected to the c-library's `system` function, and second that a pointer to the address of the part of the string which constitutes the `system` functions parameter is located 8 bytes higher on the stack than the function [43]. The latter of these two challenges is a particular issue when we consider that ASLR in Ubuntu 9 randomises the starting location of the stack. Hence the injected attack string will not be successful if it contains a hard coded stack address pointing to the start address of the buffer. To avoid the need to hardcode this it's necessary to find the address of a pointer to the start address of the buffer, i.e. the desired `system` function's parameter, within the stack and overwrite the stack such that the `system` function is located 8 bytes before it. Our vulnerable application has such a pointer in the stack frame of a previous function and so this is achievable. However, the side effect of modifying the stack in this way is that the `system` function will not be positioned over the return address for the function. The method of overcoming this, as documented by Shacham [2], involves injecting a number of iterations of a `ret` operator address between the functions expected return address and the address of the `system` function. This operator can easily be discovered in the binary of the application. Hence `ret` codes are injected onto the stack such that they overwrite the original return address and continue until 12 bytes from the pointer to the buffer. At this point the `system` function address is written onto the stack as shown in fig. 8.

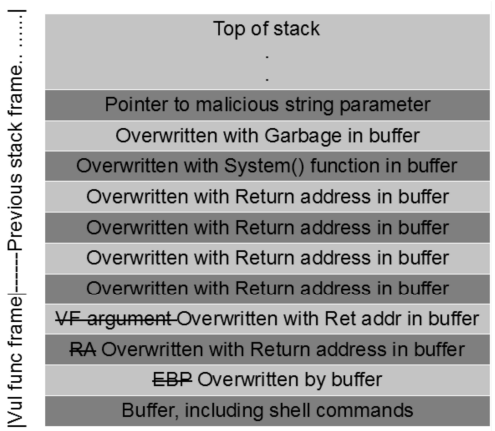


Fig. 8 Stack after malicious assault

Similar to a nop sledge which allows shell code to be slid into, this could be considered a "ret op" sledge offering a similar purpose of moving through the stack. When the vulnerable function returns it will pop the address of the `ret` operator off the stack, change the instruction pointer (`EIP`) to this address and resume execution from there. This then causes the execution of another `ret` operator and the process continues until the final `ret` in the sledge is executed and the address of the `system` function is placed in the EIP with its string parameter accurately located +8 bytes further on. Since the `system` parameter points to the address of the buffer we insert the intended `system` parameter string into the start of the buffer variable. The string in our attack is:

```
sh –c 'wget www.attack.com/rshell/rshell;
chmod +x rshell;./rshell
```

Executing this causes the victim's machine to download the malicious application *rshell*, change its permission to ensure it is executable, and then execute it.

2) Developing rshell

The application *rshell* is designed to be downloaded and executed on the victim's machine. Once executed it makes a connection to the attack machine which is set listening, in our example we used the tool Netcat [44] for this purpose. The application *rshell* makes use of the `dup2` function as follows; where `s` is the file descriptor for the connection to the attack box.

```
dup2(s, 0);
dup2(s, 1);
dup2(s, 2);
```

file descriptors 0,1 and 2 are the descriptors for stdin, stdout and stderr respectively [45]. Hence this will have the action of redirecting all input and output to and from the victim to the attack box. The net effect of this is to allow the attacker control of the victim's machine. Since this attack is initiated from the client and not the server it will mitigate protection afforded by many stateful packet inspection (SPI) firewalls and NIDS rules which trigger on "flow established to server" rules.

IV. Attack Mitigation using host based protection mechanisms

The previously outlined attack shows that both (W⊕X) and ASLR do not adequately protect against this form of attack. However, since the first documented report of this attack method, Linux distributions have undergone a changes that affect the efficiency and viability of the attack. This section will outline the affect that compiler created buffer overflow protection mechanisms and prelinking have had in this area and discuss their performance and potential for wide spread mitigation of the attack.

A. *Placing c-library functions below 0x01000000*

Some Linux distributions employ a protection patch created by the Openwall project [29] e.g. Fedora 10. This patch ensures that the positioning of the c-library functions are always below 0x1000000. This ensures that if any c-library functions address is injected onto the stack it will include a null byte (/0). Many of the dangerous non bounds checking string handling functions, which allow this attack to be possible, look for a null byte to terminate strings e.g. `strcpy`. Hence the attack outlined here, which relies on the `strcpy` function, fails when implemented in Fedora core 10. Only a fragment of the malicious string will be copied into the buffer as it terminates at the point it reaches the null byte in the `system` function address. While the Openwall Project's patch is available for most Linux distributions it is not applied by default to many of them e.g Ubuntu 9. In addition where the patch is applied a similar attack can still be crafted if the overflow is caused using a non-bounds checking function which doesn't terminate on a null byte, e.g. `recv`.

B. *Prelink*

Due to the significant number of shared libraries now deployed there is a considerable performance price in relocating these libraries during dynamic linking. Prelink speeds this up by doing it in advance, calculating address offsets for each library to ensure that during execution they will not be loaded into the same address space, and then storing these offsets in the libraries themselves. By doing this for all shared libraries and object files the time taken to start applications is reduced [45]. Unfortunately however Prelink is not compatible with ASLR since ASLR randomises the address space layout for each process on execution and thus negates the work done by Prelink. In order to address the security deficit left by disabling ASLR, Prelink randomly selects the address bases the libraries are loaded at. However this is only done when Prelink is run (this is performed every 2 weeks) [46], rather than for each execution of a process, and thus it could be viewed as less effective. Indeed this could be the reason that many Linux distributions have not enabled it by default, including Ubuntu. When considering its effect on perpetrating the act defined in this paper it is significant. Prelink cannot randomise bits 0-11 as they are used for the page offset however unlike PaX bits 28-31 can be randomised, hence 20 bits of entropy. This would increase the maximum number of guesses needed in the reconnaissance phase to 2^{20}. This would significantly increase the amount of time taken to perform the reconnaissance part of attack. Our experiment performed approximately 18.45 guesses per second, thus it would take an average of 7hrs 53 minutes to complete the reconnaissance and determine the c-library function addresses. Nevertheless once the reconnaissance is completed the address locations of the c-library functions used for all processes are known up to the time Prelink is run. Since ASLR would be disabled on any machine with Prelink enabled this signifies that the attacker can perform standard return-to-libc attacks on any buffer overflow vulnerable applications, pre-forking or otherwise. Essentially then it could be demonstrated that while prelinking would cause an increase in the initial reconnaissance time, once it has completed the system would have an increased vulnerability to return-to-libc attacks.

C. *Stack guards and SSP*

The effect of the stack guard, as outlined in section II, is demonstrated when the vulnerable application has been compiled using a GCC version with SSP enabled. To determine SSP's effect on our attack we replicated the reconnaissance phase using Ubuntu 9 which is distributed with GCC v4.3.3. When attempting to perform the reconnaissance phase of this attack under this new environment the attack failed to uncover the address of `usleep`. SSP uncovered each attempt to overflow the buffer and immediately terminated the child process thus preventing a successful overflow. The experiment was repeated on Fedora 10 using the identically compiled vulnerable application and

matching results were observed. While these results indicate that SSP is a significant tool in preventing buffer overflow attacks, a number of factors need to be considered. Firstly SSP was not introduced into GCC until the release of version 4.1 on February of 2006 [31], meaning all applications compiled using GCC prior to this will not be afforded its protection. Second, since SSP was not enabled by default until version 4.3.3 in January 2009 [31] any application compiled with GCC prior to this would be required to have the feature manually activated using the switch settings at compilation. With the onus to do this on the developer it may be reasoned that either without an understanding of merits or possibly even an awareness of the feature, this might not be performed. Further, a developer may choose to intentionally compile without SSP due to concerns over performance overhead, which has been recorded as up to 8% [11], [47]. In addition not all programs will operate correctly when compiled with SSP enabled, in particular software developed with the Gecko API [48], [49].

V. Intrusion Detection Systems Rules

The previous section argues that while the numerous host based protection mechanisms (operating system compiler and cpu) are effective in preventing brute force return-to-libc attacks their success is only assured if all the mechanisms are implemented on all systems on the network. As previously discussed this is not always feasible, realistic or desirable. An alternative approach could be to detect the attack at the network perimeter and either nullify it at that point (Intrusion Prevention), raise an alert to allow further analysis of the situation (Intrusion Detection), or to automate an action or actions to prevent further breaches (Active Response). Each of these measures would require an accurate rule to be written for an IDS or IPS to ensure that the attack is detected with minimum, ideally nil, false positive or negative responses. Sourcefire, the creators of the open source IDS Snort, suggest that the most effective way to develop a rule is to design it to trigger on the vulnerability rather than the specific exploit pattern [50] and by doing so reduce the breadth of the rule i.e. the number of rules required, and increase its precision i.e. its ability to address mutations. The alternative approach is to compare empirical traffic patterns produced while a controlled attack is taking place against the vulnerable application, with that of legitimate traffic. Unique patterns differentiating the two are thus identified and used to form a rule. While this later method is a quick and often high performance way of writing rules they often lack precision leading to false negatives when the attack data is slightly modified, or false positives while parsing legitimate traffic [51].

Through this work a detailed analysis of the vulnerability and attack method has been gained to help facilitate the production of precise, high collision rate rules. Since the attack outlined in this work requires multiple phases, both active reconnaissance and attack, it would seem prudent to prepare rules for all of these phases. Further, to increase completeness rules should be created to detect not only the attack method but also the payload, another reason for this is the observation that often zero day attacks use known payloads. By using this "component based rules" approach an attack would need to mutate on all the components, including the payload, simultaneously to avoid detection. The rules outlined in the following sections are written around an attack type rather than a specific attack, further, they have not been created with a particular traffic environment in mind either. As such they have been intentionally written with high precision and collision in mind [51]. While this reduces the speed in which the rules can be processed, it is the intention that they can be modified to make them either specific to an attack or traffic environment thus expediting their process time.

A. *Rules to detect the attack*

W $\oplus$ X and ASLR protection implementations are prevalent in most modern operating systems, thus the attack rules outlined here are created with the assumption that they are enabled. As such code injection stack based buffer overflow attacks are not implicitly discussed. Further, rule precision has been increased by focusing on the generic components of an attack that are unlikely to exist in legitimate traffic. While it is acknowledged that this presumption is dependent on the role of the systems on the internal network, the rules outlined herein would need a very specific set of non-malicious circumstances to trigger them.

1) *Rule to detect the reconnaissance attempt*

The following rule was created to detect the initial reconnaissance attempt of establishing the start location of the shared c-library:

Rule 1

```
alert tcp 192.168.10.3/32 any -> any any
(msg:"Stack smashing brute force or DOA
attack"; flow:to_client,established;
flags:R; threshold: type both, track
by_dst, count 5, seconds 5; priority: 1;
classtype:attempted-user; sid:1234567;)
```

The rule counts the number of reset messages sent from the server to the client in the time specified. In this instance 5 reset messages received in 5 seconds will result in an alert; this is deemed indicative of remote connection brute force attempt.

2) *Rules to detect malicious injection*

Rule 2

```
alert tcp 192.168.10.2/32 any -> any any
(content: "Wget"; msg:"wget request,
possible malicious code download
attempt";priority: 1;
classtype:attempted-user; sid:5234567;)
```

"wget" is an application designed to retrieve information from web servers [45] and is inherent in almost all Linux distributions. The preceding rule attempts to identify when a `wget` is attempted from the server. While it could be used for a legitimate

download such as updating software, it is considered a strong enough possibility as an attack to justify its inclusion as a rule trigger.

Rule 3

```
alert tcp any any -> 192.168.10.2/32 any
(flow:to_server,established; content: "sh
-c"; msg: "shell command sent from client,
possible
remoteattack";classtype:attempted-user;
sid:3234567;)
```

As previously discussed the attack uses a pattern of a repeated return operator address taken from the executable to manipulate the stack such that the `system` function address is placed appropriately relative to its argument which exists in a previous stack frame. Since the address of the `ret` operator will depend upon the operating system, any rule devised to look for this characteristic could not look for a hardcoded address. As such Snorts `'content'` [52] option cannot be used. Fortunately Snort rules allow the use of Perl Compatible Regular Expressions (PCRE) for matching [52]. Utilising PCRE the following rule was created:

Rule 4

```
alert tcp any any -> 193.60.151.200/24
80,443,20,25,110,143
(flow:to_server,established; pcre:
([^\x00]{4})\1; msg: "repeated words,
possible stack
overflow";classtype:attempted-user;
sid:9234567; rev:3;)
```

The PCRE option of this rule looks for a repeated concurrent 4 byte pattern which contains any character other than null byte characters i.e 0x00. Detection of repeated null bytes has to be avoided due to the standard practice of NIC drivers implementing Ethernet padding [53] using null bytes. Including these in the match is likely to lead to false positives. The \1 option in the rule is a back reference option used to allow the pattern to refer back to the results of a previous match [54], in this instance an alert is raised if four non null byte characters are identified and then followed by four more identical characters. This rule is discussed further in section 5.

3) Rules to detect the Exploit Payload

In this attack the payload has not been obfuscated and as such has identifiable Unicode text in the symbol table section of the Elf binary [45] which is stealthily downloaded via the `wget` command to the vulnerable client as part of the attack. As discussed previously this malicious payload uses the `dup2` function to redirect standard input, and standard output to a maliciously connected socket thus allowing the malevolent party to take control of the system. Considering the prospect of `dup2` being used as part of a rule, due consideration is paid to the prospect of it creating false positives. The `dup2` function is commonly used in Unix based pipes allowing 2 way communication between child and parent processes [55] . In addition it is habitually used in connection based daemons to allow a child process to redirect a pipe provided by a parent process to a file descriptor specified by the child [56]. Once redirected, the parent process may optionally close the original pipe and/or terminate whilst the child continues to use the pipe. For example, a child process may use `dup2` to redirect stderr to stdout [45]. Consequentially it could be possible that a rule based on this function could fire an alert on a legitimate upgrade of a server containing it. Nevertheless `dup2` is considered likely enough to be indicative of malicious activity and is included in the following rule:

Rule5

```
alert tcp any any -> 192.168.10.2/32
any(flow:to_client,established;content:
"dup2"; msg: "dup2 in string table,
possible remote shell
attack";classtype:attempted-user;
sid:4234567;)
```

In the string "`bin/sh`", `sh` is a symbolic link in to a shell [45]; this could be any shell variant such as *bash* or *dash*. Since it is generic it is more likely to be used in a malicious attack, as opposed to a specific shell command. A payload being downloaded to a server which contains this string in the elf string table is considered to be potentially malicious and can be identified by the following rule:

Rule 6

```
alert tcp any any -> 192.168.10.2/32 any
(flow:to_server,established; content:
"/bin/sh"; msg: "binsh request, possible
remote shell attack";classtype:attempted-
user; sid:2234567;)
```

VI. TESTING THE RULES

Implementation of the security mechanisms outlined in section IV is not universal. Not all applications are complied with the necessary protection, thus benefit could be gained by detecting brute force return-to-libc attacks. In the previous section several rules were created to match both the reconnaissance, malicious string injection and undesired download phases of the attack. These generic rules act as a blueprint for more specific ones tuned to a particular organisations traffic.

All of these rules were tested using the test bed environment outlined in section III A. Due to the need for frequent repeated testing and rule modification, the attack traffic was captured to a pcap file during an initial run of the experiment. Further iterations of the experiment were then performed by running the file through Snort while monitoring the result.

The PCRE option component of rule 5 was initially tested using RegexBuddy [57] by loading the file containing the attack data, as discussed in section III D, into it. This file was thus used within the application for the purposes of testing and modifying the PCRE option to fulfil the requirement of identifying repeated 4 byte words.

Once all the rules were firing using pcap data they were subsequently all tested again simultaneously in a real-time simulated attack as outlined in section IIId.

All the rules fired as expected with zero false negatives.

To test the results for false positives 294MB of traffic was captured from a university web server and replayed into Snort and the results monitored. Initially some of the rules needed modifying to establish zero false positives, it is these modified versions that are included in this work.

1) Detection and Performance testing and rule modification

While it is not the intention to perform detailed performance testing, some minor work in this area has been performed. In our experiment Rule 4 alerted with 4744 false positives; using the Basic Analysis and Security Engine (BASE) [58] the traffic patterns responsible for creating these alerts were examined. The alerts were being largely generated by repeated ASCII "A" characters which existed as part of the http authentication negotiation procedure [59]. It would be simple to write a PASS rule to allow http authentication traffic to pass unchecked e.g. by content checking on the string "Authorization: Negotiate". However it has been documented that the type of attack discussed here could be performed during http authentication [60] and thus this would increase the likelihood of false negatives. An alternative approach was attempted which filtered out the repeated "A" characters by extending the PCRE component i.e. `pcre:"/([^\x00]{4})\1([^\x41]{4})\2/"` However this approach was abandoned as it still caused 119 false positives, on examination these were largely due to the repetitive nature of binary values contained in image downloads. Further to this, excessive filtering dependent on a solitary traffic characteristic, offsets the intention to keep the rules generic. In addition to the false positives, the rule was also CPU intensive taking over 25% of the total processing time when reading in the test traffic. This is due to the high CPU costs inherent in using PCRE and as such it is desirable to use it after a less expensive match has prequalified the pattern [61]. As such it was combined with rule 3 as shown:

Rule7 (combining rule 3 and 4)

```
alert tcp any any -> 192.168.10.2/32 any
(flow:to_server,established; content: "sh
-c"; pcre: ([^\x00]{4})\1; msg: "shell
command sent and repeated words, possible
remote attack"; classtype:attempted-user;
sid:3234567;)
```

The new rule was tested, it did not produce any false positive or negatives and the resultant processing overhead was negligible.

Since Rule 1 did not contain a "content:" option it could not make use of the fast pattern matcher employed by Snort and thus was applied against approximately 50% of the traffic during this test. While initially this appears to be problem, further analysis shows only 2.5% of the total time spent processing the test traffic was spent in processing this rule. As such in this environment this could be deemed acceptable.

VII. Conclusions

In order to determine the efficacy of mainstream host based protection mechanisms a practical return-to-libc brute force attack was constructed and launched against a vulnerable pre-forking concurrent server. Preforking listening processes is a common way of creating HTTP, SMTP, and IMAP servers. With the vulnerable application compiled using a version of GCC predating 4.3.3, the attack was able to mitigate the protection offered by both W⊕X and ASLR inherent within the Ubuntu 9 operating system. Compiling the application using GCC with the switch `-fstack-protector-all` or compiling without this switch in a version of GCC at 4.3.3 or above, in which the option is active by default [31], would prevent the attack. This is due to the insertion of stack guards which detect undesired injection onto the stack and ASLR applied to the binary preventing predictability of determining the address of return operators within the executable.

The Openwall project suggested that the memory addresses of c-library functions should be located under 0x10000000, thus ensuring if such an address was hardcoded as part of an attack it would contain a null byte and cause the malicious string to terminate without causing a security breach.

Prelinking increases performance by re-locating shared library and object files prior to dynamic linking. It is incompatible with ASLR [46], is performed by default every fortnight and offers additional entropy when compared to Arjan van de Ven's ASLR implementation. Prelink affects the reconnaissance phase of the attack by increasing the time taken. Still, since it is incompatible with ASLR, once this phase had completed the address space for all applications would be known until Prelink is run. Thus it could be argued that Prelink increases performance at the cost of security.

Deficiencies of host based protection mechanisms were recognized and the residual threat of brute force return-to-libc attacks established. As such Snort was used to demonstrate how NIDS's can be employed to mitigate this threat. Several rules were created that when exposed to 294mb of traffic from a universities web server showed no false positives. When exposed to a simulated brute force return-to-libc attack each rule fire as expected, 100% true positive. The processing of the rules exhibited an acceptable overhead.

When considering future work; in this study we discovered repeated attack patterns in the traffic. Study into their detection via intelligent pattern matching algorithms such as the Motif tracking algorithm [62], could prove fruitful.

Further, mobile devices are becoming more sophisticated, acting as both peers and servers. A question arises; can they be hacked in a similar method to that discussed in this work? Since Android phones utilise a Linux kernel similar to that used in these experiments initiating a similar attack seems likely. If so, can this risk of attack be mitigated through NIDS?

VIII. ACKNOWLEDGEMENT

This paper is partially sponsored by the National Natural Science Foundation of China, Funding Number 60873208.

REFERENCE

1. *Browser Security: Lessons from Google Chrome.* **Reis, C, Barth, A and Pizano, C.** New York : ACM, 2009, Vol. 52(8), pp. 45-49. doi:10.1145/1536616.1536634.
2. *On the Effectiveness of Address Space Randomization.* **Shacham, Hovav, et al.** Washington : ACM, 2004. 1-58113-961-6.
3. **Sotirov, Alexander and Dowd, Mark.** Bypassing Browser Memory Protections. *The art of software security assessment.* [Online] 7th August 2008. [Cited: 25 October 2008.] taossa.com/archive/bh08sotirovdowd.pdf.
4. *Fast and Automated Generation of Attack Signatures: A Basis for Building SelfProtecting.* **Liang, Z and Seikar, R.** Tucson : ACSAC, 2005. 21st Annual Computer Security Applications Conference (ACSAC). pp. 215-224.
5. **Foster, James, et al.** *Buffer Overflow attacks.* Burlington : Syngress, 2005.
6. SANS. *The Top Cyber Security Risks.* [Online] Sans, September 2009. [Cited: 06 April 2010.] http://www.sans.org/top-cyber-security-risks/#trends.
7. **TIOBE Software.** TIOBE Programming Community Index for September 2008. *TIOBE Software.* [Online] September 2008. [Cited: 11 September 2008.] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
8. *Building Software Securely from the Ground Up.* **Ghosh, A K, Howell, C and Whittaker, J A.** Washington : IEEE SOFTWARE, 2002, Vol. 19(1). pp. 14-16. MS.2002.976936.
9. **Schmidt, C and T, Darby.** The What, Why, and How of the 1988 Internet Worm. *The What, Why, and How of the 1988 Internet Worm.* [Online] snowplow.org, July 2001. [Cited: 6 April 2010.] http://www.snowplow.org/tom/worm/worm.html.
10. **Cowan, C.** Buffer Overflow Attacks. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.* [Online] 9 December 1997. [Cited: 1 October 2008.] http://www.usenix.org/publications/library/proceedings/sec98/full_papers/cowan/cowan_html/node3.html.
11. **Etoh, H.** Evaluation. *GCC extension for protecting applications from stack-smashing attacks* . [Online] IBM, 11 August 2000. [Cited: 27 January 2010.] http://www.trl.ibm.com/projects/security/ssp/node5.html.
12. *Death, Taxes and Imperfect Software: Surviving the Inevitable.* **Crispin Cowan, Carlton Pu, and Heather Hinton.** Charlottesville : Association for Computing Machinary, 1998. Proceedings of the New Security Paradigms Workshop.
13. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.* **Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole.** Washington : IEEE Computer Society, 2000. DARPA Information Survivability Conference & Exposition - Volume 2. pp. 119-129.
14. **Bradley, Tony.** Introduction to Intrusion Detection Systems. *About.com.* [Online] 2nd April 2008. [Cited: 09 January 2009.] http://netsecurity.about.com/cs/hackertools/a/aa030504.htm.
15. *A Neuro-genetic Based Short-term Forecasting Framework for Network Intrusion Prediction System, International Journal of AUTOMATION AND COMPUTING.* **Sindhu, S,S,S, et al.** 4, Dordrecht : Springerlink, 2009, Vol. 6(4). pp. 319-363.1751-8520.
16. *Network-Level Polymorphic Shellcode Detection Using Emulation.* **Polychronakis, Michalis, Anagnostakis, Kostas G and Markatos, Evangelos P.** Paris : Springer Paris, 2006, Vol. 2(4). pp. 257-274 1772-9890.
17. *A polymorphic Shellcode Detection Mechanism in the Network.* **Huang, Hsiang-Lun, et al.** Suzhou : ACM, 2007. 978-1-59593-757-5/07/0006.
18. **Lippmann, R, Webster, S and Stetson, D.** The Effect of Identifying Vulnerabilities and Patching Software on the Utility of Network Intrusion Detection. [book auth.] A Wespi, G Vigna and L Deri. *Recent Advances in Inrusion Detection.* Berlin / Heidelberg : Springer , 2002.
19. Rule Performance Part One: Content Matches. *VRT.* [Online] Sourcefire, 8 July 2009. [Cited: 6 April 2010.] http://vrt-sourcefire.blogspot.com/2009/07/rule-performance-part-one-content.html.
20. **Aleph1.** Smashing The stack for fun and profit. *Phrack Magazine.* [Online] 8 September 1996. [Cited: 08 September 2008.] http://www.phrack.org/issues.html?issue=49&id=14#article.
21. **Haendel, Lars.** The function pointer tutorials. *www.newty.de.* [Online] 6 January 2005. [Cited: 13 September 2008.] http://www.newty.de/fpt/intro.html#what.
22. **Etoh, Hiroaki.** Stack Protection Systems: (propolice, StackGuard, XP SP2). *pacsec.jp.* [Online] 2004. [Cited: 13 September 2008.] pacsec.jp/psj04/psj04-hiroaki-e.ppt.
23. **Schildt, Herbert.** *C++ A beginners Guide* . Maidenhead : McGraw-Hill Professional, 2003.
24. **Sanders, Chris.** Buffer Overflows, Data Execution Prevention, and You. *WindowSecurity.com.* [Online] WindowSecurity.com, 28 October 2009. [Cited: 6 April 2010.] http://www.windowsecurity.com/articles/Buffer-Overflows-Data-Execution-Prevention-You.htm.
25. **van de Ven, Arjan.** New Security Enhancements in Red Hat Enterprise Linux v.3, update 3. Raleigh, North Carolina, USA : Red Hat, August 2004.
26. **Whitehouse, Ollie.** *An Analysis of Address Space Layout Randomization on Windows Vista.* Cupertino : Symantec, 2007.
27. **appsec.ch.** Bypassing Windows Vista's Address Space Layout Randomization. Lösliweg, Felsberg, Switzerland : skillTube.com, 2007.
28. *Secure and Practical Defense Against Code-Injection Attacks using Software Dynamic Translation.* **Hu, W, et al.** New York : s.n., 2006. ACM/Usenix International Conference On Virtual Execution Environments . pp. 2-12.
29. Linux kernel patch from the Openwall Project. *Openwall Project.* [Online] September 2002. [Cited: 06 December 2009.] http://www.openwall.com/linux/.
30. **Lacroix, P and Desharnais, J.** *Buffer Overflow Vulnerabilities in C and C + +.* s.l. : Unpublished report, 2008.
31. **GCC steering Committe.** http://gcc.gnu.org/releases.html. *GCC, the GNU Compiler Collection* . [Online] 6 April 2010. [Cited: 6 April 2010.] http://gcc.gnu.org/releases.html.
32. **skape.** Preventing the exploitation of SEH overwrites. *Uninformed.* [Online] September 2006. [Cited: 28 September 2008.] www.uninformed.org/?v=5&a=2&t=pdf.
33. **Security Focus.** Oracle 9I Application Server PL/SQL Apache Module Buffer Overflow Vulnerability. *SecurityFocus.* [Online] SecurityFocus, 11 July 2009. [Cited: 6 April 2010.] http://www.securityfocus.com/bid/3726/discuss.
34. **Stevens, R S, Fenner, B and Rudoff, Andrew M.** *Unix Network Programming.* Boston : Pearson Education, Inc., 2003.

35. *Defeating PaX ASLR protection.* **Durden, T.** 59, s.l. : Phrack, 2002, Vol. 12.
36. **vmware.** Workstation 7. *vmware.* [Online] vmware, 2010. [Cited: 6 April 2010.] www.vmware.com/workstation.
37. **Wireshark.** Wireshark. *Wireshark.* [Online] [Cited: 14 April 2010.] http://www.wireshark.org/.
38. **Sourcefire.** Snort. *Snort.* [Online] Sourcefire. [Cited: 14 April 2010.] http://www.snort.org/.
39. *The advanced return-into-lib(c) exploits.* **Nergal.** 58, s.l. : Phrack Inc, 2001, Vol. 11.
40. *Explotation for Phun and Profit.* **Bucko, C.** s.l. : HACKPL Security Lab.
41. **van Riel, R and Feng, S.** Documentation for /proc/sys/kernel. *The Linux Kernel Archives.* [Online] 2009. [Cited: 06 April 2010.] http://www.kernel.org/doc/Documentation/sysctl/kernel.txt.
42. **the PaX Team.** Documentation for the PaX project. *Homepage of The PaX Team.* [Online] [Cited: 6 April 2010.] http://pax.grsecurity.net/docs/index.html.
43. **Rash, Michael, et al.** *Intrusion Prevention and Active Response.* Rockland : Syngress Publishing, Inc., 2005.
44. **The GNU Netcat Project.** The GNU Netcat Project. *The GNU Netcat Project.* [Online] [Cited: 14 April 2010.] http://netcat.sourceforge.net/.
45. **LinuxManPages.com.** LinuxManPages.com. *LinuxManPages.com.* [Online] [Cited: 6 April 2010.] http://linuxmanpages.com/.
46. **Moser, J R.** Prelink and address space randomization. *LWN.net.* [Online] 5 July 2006. [Cited: 6 April 2010.] http://lwn.net/Articles/190139/.
47. *Buffer Overflows:Attacks and Defenses for the Vulnerability of the Decade.* **Cowan, C, Wagle, P and Calton, P.** Pennsylvania : IEEE, Attacks and Defenses for the Vulnerability of the Decade. 0-7695-0490-6/99.
48. **Mozilla.** Mozilla wiki. *Mozilla wiki.* [Online] Mozilla, 26 January 2010. [Cited: 27 January 2010.] https://wiki.mozilla.org/Gecko:Home_Page.
49. **Alioth Project.** Stack Smash Protection. *Debian SbD.* [Online] Alioth Project. [Cited: 27th January 2010.] http://d-sbd.alioth.debian.org/www/?page=ssp.
50. **Sourcefire Vulnerabilty Research Team.** Sourcefire Vulnerability Team. *Sourcefire Vulnerability Team.* s.l. : Sorcefire, 2006.
51. *Writing detection signatures.* **Jordan, Christopher.** 6, Berkeley : USENIX;login:, 2005, Vol. 30.
52. **The Snort Project.** Snort Users Manual. *Snort Users Manual.* s.l. : Snort, 2009.
53. **IEEE Computer Society.** Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. *Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.* New York : IEEE, 2008. ISBN 973-07381-5796-2 STD95819.
54. **Hazel, P.** PCRE - Perl Compatible Regular Expressions - Man pages. *PCRE - Perl Compatible Regular Expressions.* [Online] 3 January 2010. [Cited: 6 April 2010.] http://www.pcre.org/pcre.txt.
55. **Friedl, Steve.** Mapping UNIX pipe descriptors to stdin and stdout in C . *Steve Friedl's Unixwiz.net Tech Tips.* [Online] [Cited: 3 February 2010.] http://unixwiz.net/techtips/remap-pipe-fds.html.
56. **Leffler, Samuel J, et al.** An Advanced 4.4BSD Interprocess Communication Tutorial. [Online] [Cited: 03 February 2010.] http://docs.freebsd.org/44doc/psd/21.ipc/paper.pdf.
57. **Goyvaerts, Jan.** JGsoft. *Learn, Create, Understand, Test, Use and Save Regular Expressions with RegexBuddy.* [Online] Just Great Software Co. Ltd, 11 March 2010. [Cited: 6 April 2010.] http://www.regexbuddy.com/.
58. Basic Analysis and Security Engine . *Basic Analysis and Security Engine* . [Online] Basic Analysis and Security Engine , 28 May 2009. [Cited: 6 April 2010.] http://base.secureideas.net.
59. Davenport WebDAV-SMB Gateway. *The NTLM Authentication Protocol and Security Support Provider.* [Online] GNU. [Cited: 6 April 2010.] http://davenport.sourceforge.net/ntlm.html#ntlmHttpAuthentication.
60. Vulnerability Note VU#878603. *US-CERT United States Computer Emergency Readines Team.* [Online] US Department of Homeland Security, 15 March 2002. [Cited: 6 April 2010.] http://www.kb.cert.org/vuls/id/878603.
61. **Baker A, R, et al.** *Snort IDS and IPS Toolkit.* Burlington : Syngress, 2007. 1-59749-099-7.
62. *The Motif Tracking Algorithm, International Journal of AUTOMATION AND COMPUTING.* **Wilson, W, Birkin, P and Aickelin, U.** 1, Dordrecht : SpringerLink, 2008, Vol. 5. pp. 32-44. 1751-8520.

Authors Biography

David J Day, BSc, MSc in computing systems and computer networks, he is a Senior Lecturer in Computing and a Teaching Fellow for the Faculty of Business Computing and Law at the University of Derby. He is a PhD candidate in networking systems and his research interests include computing system intrusion detection and prevention, mobile device security and computer network management.

Zhengxu Zhao, BSc, MSc, PhD in computing science and technology, Professor and Chair in Applied Computing at the University of Derby from 1995 and 2008 and holds a DSc from Derby for his research work in information technology and scientific visualization. He is currently Professor and Dean of Faculty of Information Science and Technology at the Shijiazhuang Tiedao University, China. His research interests include virtual reality systems, scientific visualization, and information organization.