2010 Fourth International Conference on Digital Society

Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems

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

Zhengxu Zhao
School of Computing and
Information Engineering
Shijiazhuang Tiedao Institute
Shijiazhuang Hebei, China
Zhaozx@sjzri.edu.cn

Minhua Ma
School of Computing
University of Derby
Derby, UK
m.ma@derby.ac.uk

Abstract— **There has been a significant amount of research recently into methods of protecting systems from buffer overflow attacks by detecting stack injected shell code. The majority of the research focuses on developing algorithms or signatures for detecting polymorphic and metamorphic payloads. However much of this problem has already been solved through the mainstream use of host based protection mechanisms e.g. Data Execution Prevention (DEP) and Address Space Randomization (ASLR). Many hackers are now using the more inventive attack methods e.g., return-to-libc, which do not inject shell code onto the stack and thus evade DEP and common shell code detection mechanisms. The purpose of this work is to propose a series of generic signatures that could be used to detect network born return-to-libc attacks. To this end we outline how we performed a return-to-libc network based attack, which bypasses DEP and common IDS signatures, before suggesting an efficient signature for detection of similar return-to-libc attacks.**

Keywords-Buffer;Overflow;IDS;Stack;Return-to-libc;Signature;Exploit;Vulnerability;Protection;Network; Attack;Rules

I. INTRODUCTION

Buffer overflow vulnerabilities exist due to weak programming techniques which employ unsafe string handling functions and inadequate bounds checking. Buffer overflow bugs are extremely widespread; they are responsible for nearly all Internet worms [1] and are the most commonly reported software vulnerabilities [2].

Applications which are created with these vulnerabilities are open to malevolent attacks allowing a malicious party to redirect a programs flow of execution and gain access to the victim's system. Access is gained by compromising the memory regions of either the stack or heap. In the case of the stack typically a buffer declared locally in a vulnerable function is flooded overwriting either the functions return address, function pointers arguments or its exception handler on the stack. On returning from the vulnerable function, calling a compromised function pointer or throwing an exception (respectively) the malicious perpetrator will redirect content flow. Conversely, heap based compromises classically overwrite the boundaries of a memory chunk such that a "fake" chunk is crafted. The design of this chunk takes advantage of the unlink routine during coalescence to redirect program flow to an address of the attackers choosing when the memory is freed. Regardless of the exploit mechanism once the attacker has gained control they can use the system for a variety of malignant purposes, e.g. introducing root kits and other malware or malicious reconnaissance.

Since the earliest recorded buffer overflow worm was launched, the Morris worm in 1988 [3], numerous mechanisms for protecting systems against buffer overflow attacks have been implemented. While host based methods have made significant progress in securing systems with techniques such as Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) raising the bar significantly, they have not proved impenetrable. The limitations of host based protection systems have prompted security professionals to consider using Network Intrusion Detection System (NIDS) signatures to help thwart buffer overflow attacks; however hackers are seeking to find inventive ways to evade these too, e.g. rendering their payload stealthy by applying polymorphic or other obfuscation techniques. As a result much work has been done to tune NIDS buffer overflow detection signatures to identify obfuscated injected code. However, while the majority of attack methods inject code into vulnerable applications (code injection attacks) variant attack methods are now being constructed which do not require code to be injected and instead rely on code already loaded into the system at run time to perform the attack (existing code attacks). By avoiding using injected code they can render the majority of NIDS signatures ineffective.

The remainder of this paper will be constructed as follows. Section II will discuss the threat posed by traditional code injection buffer overflow attacks. Section III addresses the efficacy and limitations of popular NIDS and host based protection mechanisms in countering buffer overflow attacks. Section IV will outline the basic operation of return-to-libc attacks and how it's used to bypass DEP. This section outlines a network based return-to-libc simulation we constructed as part of the research. In section V we propose how to protect against return-to-libc network attacks using effective, efficient and novel IDS signatures. Finally, in section VI, we offer our

978-0-7695-3953-9/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDS.2010.37

conclusions and communicate the direction of our research for forthcoming dissemination.

II. Code Injection Buffer Overflow Attacks

Consider the stack layout (Figure 1) after a vulnerable function has been subjected to a typical code injection buffer overflow attack. Please note NOP stands for no operation and is an operator that does not perform any operation, the processor will simply move to the next instruction in sequence eliminating the need to know the exact location of the shell code on the stack. EBP is the pointer register often called the frame pointer which is used as an offset to locate the other components on the stack frame.

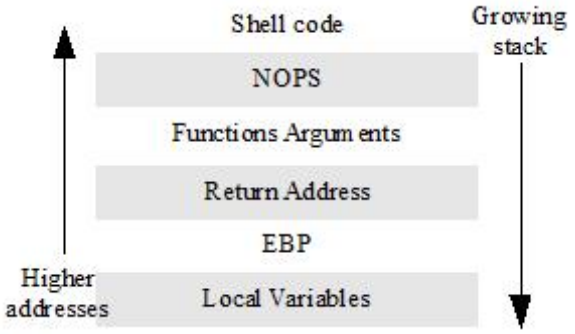

Figure 1. Stack state after a traditional attack

Here the memory reserved for the buffer is flooded such that functions return address is overwritten, nop operators (nop sledge) are inserted and shell code is added to the stack. The overwritten return address contains the address close shell code. While the hacker would prefer to overwrite the return address with the exact address of the shell code this is often difficult to predict. When the function exists it will be redirected to a point close to the shell code whereby it move through each nop instructions on the sledge until it ultimately executes the shell code.

III. Protecting Against Buffer Overflow Attacks

There are two main methods of detecting and/or preventing buffer overflow attacks, these are either over the network or on the host machine. The following two sections discuss both Network Intrusion Detection and Prevention Systems (NIDS/NIPS) and host based CPU, operating system and compiler based protection mechanisms.

A. Network Based Intrusion Detection and Prevention Systems

Network intrusion detection and prevention systems can offer defense during the unavoidable periods where systems are vulnerable by alerting on (intrusion detection) or blocking (intrusion prevention) traffic streams whose characteristics match that of known attacks. There are a couple of widespread ways in which an IDS system can be used to detect buffer overflow attacks. Most commonly manually crafted individual signatures can be written for each. Disassembling vulnerable network applications to determine what constitutes illegitimate traffic for that application and using this to form the signature. Unfortunately however these signatures do not offer any protection against zero day attacks i.e. attacks upon vulnerabilities which are yet to be uncovered by the protection communities. One popular way to help counter zero-day attacks is for IDS signatures to match against known shell code patterns [4]. Popular methods of doing this involve examining the traffic for no operation (nop) operators used to create a no-op sledge [5]. This often proves effective as many hackers do don't have the ability or inclination to customize their shell code to avoid this. In addition to shellcode patterns being detected by IDSs hackers face other challenges, for example shellcode containing null bytes will terminate prematurely (prior to the malicious payload being delivered) when using functions such as strcpy and strcat. As such it is the aim of the hacker to remove terminating bytes from the shellcode. This fact is understood by IDS signature writers who, concerned with false positives and efficacy, often immediately pass any traffic which contains terminating bytes.

Inventive hackers can bypass both simple IDS shell code pattern detection and null byte terminators by using shell code obfuscation techniques such as shell code encryption and polymorphism. Encryption works by encrypting the shellcode before injecting and including a decoder in the payload to decrypt it on execution. One simple method of encryption is to use an arbitrary value to XOR each byte of the shellcode [6] and once the payload is executed a decoder will translate the encrypted code back into an executable commands [7]. To create polymorphic shell code, using the previous example, the encryption value can be randomised at execution. This is a simple example, there are a number of other, more inventive, methods of encoding shellcode and the whole process has been simplified through the use of polymorphic shellcode generators such as the ones included in the Metasploit Framework [8], ADMutate or Tapion [9]. In an effort to counter this new threat there has been significant research into polymorphic shellcode detection, much of which involves efforts to uncover the decode engine as it is transmitted over the network, however intelligent design is often used to ensure the code used in these encryption algorithms changes with each execution (metamorphism) [10]. As researchers strive to create more intelligent signatures to identify polymorphic code they increase the chance of the IDS providing a high rate of false positives. As a result of this it could be argued that the most effective tool to combat shellcode injection attacks are the host based protection mechanisms such as DEP. By ensuring the DEP is active for all

applications code injection attacks, obfuscated or otherwise, should not be a concern.

B. Host Based Protection Mechanisms

Host based protection relies principally on using safe compiler options which render the compiled code less susceptible to buffer overflow attack. The most prominent host based protection mechanisms are stack based buffer overrun detection, safe structured exception handling, Data Execution Prevention (DEP) (sometimes referred to as W X) and Address Space Layout Randomisation (ASLR).

Stack based buffer overrun detection operates by introducing a canary value onto the stack prior to the return address during function compilation. Code is emitted such that, during the epilog, the canary is checked to see if it is the same as the value originally placed on the stack, if it's not then it's assumed an overflow has occurred and execution terminates [11]. Additional protection involves compiling using a safe structured exception handling option which results in an image being created that includes a table of safe structured exception handler's [12]. When an exception occurs, the exception handler's addresses are validated against the safe exception handlers table and if the address isn't contained in it then the application terminates. DEP involves preventing sections of the stack and the heap from containing executable code, e.g. shellcode, by marking these areas as Writable XOR eXecutable (W X) i.e. they can be written to, or executed from, but not both. ASLR is designed to prevent attacks which use "existing code" such as functions in the c-library of a system (return-to-libc) instead of injecting their own code, and thus circumnavigating DEP. These outlined defence mechanisms have evolved as each of the previous mechanisms have been thwarted using increasingly innovative attack approaches. As a result of this evolution to offer effective protection it is the application of all the individual protection methods combined that prove a formidable defence. Essentially this means every binary in the system must have been compiled with multiple safe compiler options from the appropriate compilers and be running on a compatible platform to support the safeguards they provide. This would seem an unlikely scenario for the majority of organisations as only later operating systems and CPUs support them and many popular software applications do not, e.g. Firefox 2 and Internet Explorer 7 do not support ASLR and opt out of DEP by default. Unfortunately during the time it takes to ensure all systems are making use of these protection mechanisms history indicates that hackers will have devised inventive methods of bypassing them.

IV. Return-to-libc Buffer Overflow Attacks

Existing code attacks e.g. return-to-libc have been devised with a view to bypassing the host based protection mechanisms DEP. This is achieved by avoiding putting shellcode onto the stack. With DEP enabled shellcode can not be executed since those areas of the stack are marked as non executable.

One of the ways of circumnavigating this protection mechanism is to use a technique commonly referred to as return-to-libc. This style of attack needs neither an executable stack or shellcode but instead causes the vulnerable program to jump to some existing code in the C library e.g. the `system()` function [13]. The stack layout can be crafted such that the address of the system function plus its argument is placed on the stack, see figure 2.

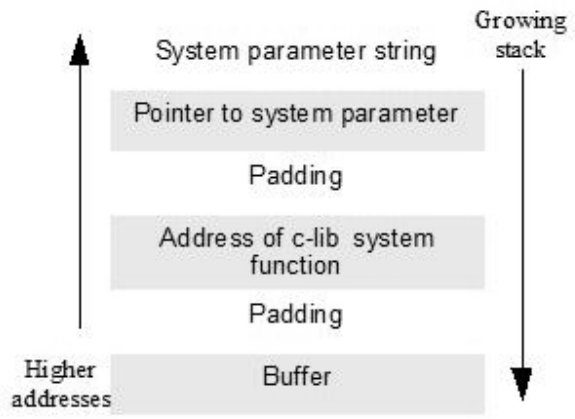

Figure 2. Stack of stack after a return-to-libc attack

A. Simulation of a Network Based Return-to-libc Attack

The following attack was inspired by Shacham [14] and does not use a NOP sledge, generates traffic including terminating characters and does not inject code on to the stack, thus avoiding IDS detection signatures and DEP. The outlined attack will not bypass the host based protection ASLR, however it should be noted that ASLR has only been introduced on windows versions Server 2008 and Vista or later, and even then only on selected dll's and Microsoft proprietary executables. Early operating systems such as Windows XP and 2003 server still remain vulnerable to this method of attack. At the time of writing Windows XP is the most common client operating system with a market share of 67.1% compared with Vista's 17.7% [15]. For the purpose of the experiment address space randomization was turned off on Fedora Core 9 box.

B. Environment for Attack Simulation

The environment used to create and detect the return-to-libc attack is shown in figure 3.

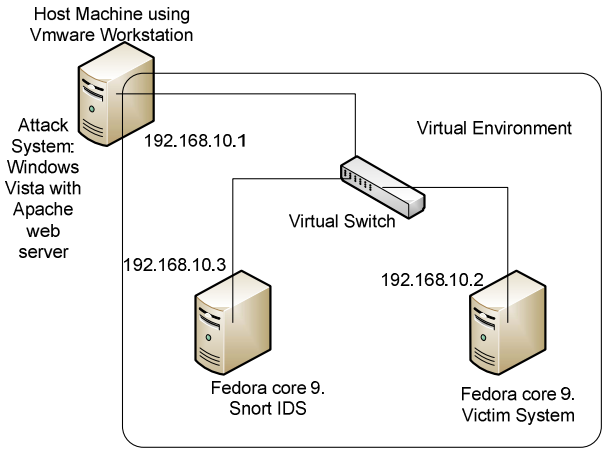

Figure 3. Attack Simulation Environment

Virtual machines (VM) were used to facilitate a more efficient and contained environment. The host machine contains the entire virtual environment facilitated via VMware workstation v6.5.2. The victim VM configured with Linux Fedora core 9 with DEP activated and ASLR turned off. Its IP tables (Linux host firewall) is active with port 23456 open to allow communication with a simple, vulnerable, network application. The attack system does not use a VM but is performed from the host. It holds an exploit payload on its apache web server and has a command shell with Netcat [16] listening on port 999. The Snort box is used to monitor and signature match traffic on the local network

C. *Details of the Attack*

The vulnerability exists as a susceptible function in the victim's network application can receive network data through a previously established address/port structure without performing bounds checking. The overflow exploit is caused by the attack system pushing through more data then the buffer in the victim's application has been declared to receive. This is achieved by piping the contents of a specially crafted file from the attack machine to the victim's application via Netcat. When the data is received by the applications vulnerable function the return address on the stack frame is overwritten with the address of the c-library `system()` function. 8 bytes on the address of the systems parameter is also pushed onto the stack, followed by the parameter string itself. When the vulnerable function returns it transfers the flow of execution to the `system()` function, see figure 2.

The system function would be expecting the top of the stack to contain its return address and the subsequent stack value to contain its argument. As can be seen the overflow has been constructed such that the `system()` function's argument is located in its desired position and will be executed as intended by the attack.

The system() function's parameter is constructed as shown below

```
"sh -c 'wget
192.168.10.1/rshell/rshell;chmod +x
rshell; ./rshell'"
```

When the vulnerable function returns the `system()` function will be executed with this parameter. Subsequently this causes Rshell to be downloaded from the attacking machine, it permissions to be changed to make it executable and is then executed.

Rshell makes a connection from the victim's machine to a listening shell (using Netcat) on the attacker's machine on port 999. Once the connection is made it uses the `dup2()` function to direct the standard input/output to the connected socket. This gives the attacker access to the shell. Since the whole attack is performed without the need to place shellcode on the stack it avoids DEP.

V. RULES TO DETECT BUFFER OVERFLOW ATTACKS

Signatures can be written to check against the vulnerability, the exploit or some generic mechanism of the attack type, for example NOP sleds. 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 [17]. That argument has merits as it will increase the precision and reduce the breadth; however it requires detailed knowledge of the vulnerability. The challenge, however, is not only to prevent the precise discussed attack on this detailed application but to try and generate some generic triggers that could help identify a nonspecific or mutated form of the attack. As such a detailed knowledge of the vulnerability isn't applicable. It should also be considered that a low breadth signature (single signature) which attempts to identify multiple patterns in both the exploit and the signature (more complete) has less precision. While this is reported to reduce false positives and negatives it is more likely to fail when the attack mutates. It has been suggested [18] that less emphasis should be placed on false positives and negatives such that the accent is moved to examining various components of both the vulnerability and the exploit using multiple signatures to reach completeness. Adding breadth in this way will increase the number of false positives and negatives but will also offer protection against zero day attacks and mutated attacks.

A. *Proposed Return-to-Libc Generic Signatures*

With the widespread implementation of DEP and other W E (Write XOR Execute) implementations we have focused on creating our signature rules with the assumption this protection is in place, hence we will not be addressing code injection variants, obfuscated or otherwise.

When considering writing rules to detect this type return-to-libc attack we have endeavoured to consider the generic aspects of each component (exploit and the payload) of the signature.

B. Deriving Signature Rules for the Exploit Payload

In our simulation the payload did not include obfuscated shell code and thus includes the string "`bin/sh`". In addition it includes the function system, used to initiate the shell, and the `dup2()` function used to redirect `stdin` and `stdout` streams allowing the attacker control. We consider the presence of any of these characteristics to be classified as suspicious with their presence over the network unlikely to be benign in the overwhelming majority of instances. While a complete single signature rule which only alerts on all the aforementioned criteria being met will produce fewer false positives it is also more likely to miss mutated payloads. We conclude that it is more robust to alert if any of these individual elements is present. It should also be considered that it is straightforward to substitute the system call in the payload for the `execve` function to achieve the same result. As such, in addition to the criteria already identified, we propose the addition of a signature rule to alert on the entire family of `exec` functions. Detection of all of the identified functions is straightforward as they exist as ascii characters within the Executable and Linkable (ELF) files symbol table, this can be easily matched and alerted on by the IDS during the files download from the attacking machine to the victim.

Detection of the payload as described above should only be considered as partly addressing the attack. As previously mentioned encrypted payloads are becoming increasing more popular, and it is possible though non trivial techniques to obfuscate this payload using polymorphic and/or metamorphic methods. Hence as a multi-pronged defence we also consider the exploit.

C. Deriving Signature Rules for the Exploit Mechanism

By exploiting the vulnerability the system function parameter `"sh -c 'wget 192.168.10.1/rshell/rshell;chmod +x rshell; ./rshell'"` and the address of the `system()` function (0060A8d0) are pushed onto the stack. When considering these for signature rules the following elements are the least likely to be benign "`sh -c`", "`wget`", "`chmod`" and the system address. However a specific system address would only be useful when monitoring for an attack on an identical Linux distribution as ours, as such to keep the signature rules generic it is not suggested as a signature rule. With this is mind the following rules were derived.

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

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

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;)

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;)
```

D. Testing and Analysis

The rules were tested by performing the attack as outlined previously, with Snort IDS monitoring the network as laid out in figure 3. All the rules fired as expected.

In order to test for false positives, 250 MB of network traffic was collected from one of the busiest network servers at a university using Wireshark [19]. A custom written application [20] was used which allowed the destination addresses of traffic in the pcap file to be adjusted. Using this option the destination address of the traffic was changed to 192.168.10.2, i.e. that of the victim server. The traffic was then replayed, from the host machine, over the network, in real-time, over a 3 hour period. No alerts were observed and hence no false positives were generated.

VI. Conclusions and Future Work

We demonstrated, through a practical return-to-libc attack simulation, the common techniques used to bypass the host based protection mechanism DEP on a Linux operating system. The attack is largely generic and could, with some minor modification, be ported to work on most operating systems including Microsoft Windows XP and Server 2003, which could be considered more vulnerable due to a lack of address space randomization protection. The demonstration served to highlight the common elements which often constitute a return-to-libc network based attack. We focused on these identified elements to suggest which aspects should be used to form signature rules to detect both this specific attack and its derivatives. Basic rules were created based on our findings and these was shown to effectively detect this attack type with zero false positives. Our findings suggest that for completeness signature rules should be created that consider both the exploit string and its subsequent payload executable. In addition to improve precision and recall more generic elements should be used to trigger the rules `namely "sh -c", "wget", "chmod"` from the exploit string and

`dup2()`, `system()` and `exec()` functions from the payload.

As previously mentioned the attack we simulated will not bypass ASLR. However ASLR and DEP when used together are not impenetrable and researchers have shown how it can be bypassed [14], [21]. In particular Shacham [14] demonstrated how ASLR can be brute force attacked using a return-to-libc derivative which guesses the libc text segment offset providing increased certainty for the location of all c library functions. While it is likely that our exploit signature rules would pick up many of these brute forces variant attack types it is recognized that the detection rate could be improved using mechanisms to detect recurring libc address guesses. Research has already begun into how to mimic a similar brute force attack type and detect it practically using the Snort IDS.

ACKNOWLEDGEMENT

The authors would like to thank Dave Voorhis for his advice concerning the logistics of the attack simulation.

REFERENCES

[1] Fast and automated generation of attack signatures a basis for building self-protecting servers. Liang, Zhenkai and Sekar, R. Alexandria : ACM/IEEE, 2005. Proceedings of the 12th ACM conference on Computer and communications security. pp. 213-222.

[2] Foster, James, et al. Buffer Overflow attacks. Burlington : Syngress, 2005.

[3] Schmidt, Charles and Darby, Tom. The What, Why, and How of the 1988 Internet Worm. The What, Why, and How of the 1988 Internet Worm. [Online] July 2001. [Cited: 11 January 2009.] http://snowplow.org/tom/worm/worm.html.

[4] Schneider, Avraham Moshe. Alphanumeric Shellcode Encoding and Detection. Insecure.org. [Online] 4 August 2008. [Cited: 24 July 2009.] http://seclists.org/fulldisclosure/2008/Aug/0023.html.

[5] Foster, James, C and Price, Mike. Sockets, Shellcode, Porting and Coding. Rockland : Syngress, 2005.

[6] A polymorphic Shellcode Detection Mechanism in the Network. Huang, Hsiang-Lun, et al. Suzhou : ACM, 2007. 978-1-59593-757-5/07/0006.

[7] Chong, SK. History and Advances in Windows Shellcode. Phrack. [Online] 22 June 2004. [Cited: 27 July 2009.] http://www.phrack.com/issues.html?issue=62&id=7.

[8] The Metasploit Project. The Metasploit Project. [Online] [Cited: 27 July 2009.] http://www.metasploit.com/.

[9] Bania, Piotr. Tapion Project. www.piotrbania.com. [Online] 16 9 2005. [Cited: 1 08 2009.] http://pb.specialised.info/all/tapion/.

[10] Network-Level Polymorphic Shellcode Detection Using Emulation. Polychronakis, Michalis, Anagnostakis, Kostas G and Markatos, Evangelos P. 4, Paris : Springer Paris, 2006, Vol. 2. 1772-9890.

[11] Rash, Michael, et al. Intrusion Prevention and Active Response. Rockland : Syngress Publishing, Inc., 2005.

[12] Microsoft. /SAFESEH (Image has Safe Exception Handlers). MSDN Virtual C++ Developer Center. [Online] 2008. [Cited: 07 December 2008.] http://msdn.microsoft.com/en-us/library/9a89h429(VS.80).aspx.

[13] Wenliang, Du. Return-to-libc Attack Lab. Syracuse University - Department of Electrical Engineering and Computer Science. [Online] 2006. [Cited: 13 October 2008.] http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Return_to_libc/.

[14] On the Effectiveness of Address Space Randomization. Shacham, Hovav, et al. Washington : ACM, 2004. 1-58113-961-6.

[15] OS Platform statistics. W3Schools.com. [Online] [Cited: 5 August 2009.] http://www.w3schools.com/browsers/browsers_os.asp.

[16] Armstrong, Thomas. Netcat [software], 2004.

[17] Olney, Matthew. Writting effective rules, part I. [Webinar] Columbia : Sourcefire, 2008.

[18] Writing detection signatures. Jordan, Christopher. 6, Berkeley : USENIX;login:, 2005, Vol. 30.

[19] Combs, Gerald. Wireshark [software], 2009.

[20] Packet generators work as a stress test but are not very successful at recreating traffic in 'real-time'. Can this area be developed further to recreate a true reflection of 'regular usage' of a host, and regenerate that usage so as to simulate 'real-time traffic' on the network? Raynor, M : Unpublished undergraduate thesis, 2009.

[21] Sotirov, Alexander and Dowd, Mark. Bypassing Browser Memory Protections. In *Black Hat*, 2008.