AN EXAMINATION OF THE GENERIC MEMORY CORRUPTION EXPLOIT PREVENTION MECHANISMS ON APPLE'S LEOPARD OPERATING SYSTEM

Haroon Meer

SensePost, Rhodes University

haroon@sensepost.com

AN EXAMINATION OF THE GENERIC MEMORY CORRUPTION EXPLOIT PREVENTION MECHANISMS ON APPLE'S LEOPARD OPERATING SYSTEM

ABSTRACT

The Win32 platform has long been the whipping boy of memory corruption attacks and malware, which has forced Microsoft into implementing aggressive anti-exploitation mechanisms into their newer Operating Systems. Apple's Mac OS X (Leopard) has had a much smoother run, both in the media, and in terms of high profile attacks and the reason for this is less clear.

In light of Apple's increased market-share, a comparison between Microsoft's defences and Apple's defences is required as the number of anti-exploitation techniques increases with time. In order to produce a side-by-side comparison, an overview of memory corruption attacks is provided and the common generic anti-exploitation techniques for these attacks are enumerated and described. For each operating system, the quality and effective of each implemented defence is evaluated.

The results of the study show that Leopard trails Windows Vista in both the number of defences, as well as the quality and effectiveness of the defences that are implemented.

KEY WORDS

exploit memory corruption stack heap shellcode overflow ret-2-libc

1 INTRODUCTION

This paper will cover the basics of memory corruption exploits, and will then examine how Microsoft Windows Vista and Apple MacOS X Leopard combat these attacks in their default state. The intention is to examine how Apple’s Leopard measures up against the automatic exploit mitigations built into Vista.

In the interest of brevity and in order to remain focused, this paper will exclude comparisons between the built in firewalls, sandboxing capabilities or attacks that are not directly related to the execution of arbitrary code through memory corruption attacks. Discussion relating to the comparison of these other security features can be found in presentations and papers delivered by developers from Microsoft and Apple respectively [1, 2].

The remainder of the paper is structured as follows: Section 2 provides background on memory corruption attacks, Section 3 details and describes generic defences against memory corruption as well as attacks that bypass the defences, a comparison and analysis is provided in Section 4 and we conclude in Section 5.

2 MEMORY CORRUPTION ATTACKS

Memory corruption attacks have been publicly discussed, at least since the 1988 Morris worm, which exploited a buffer overflow vulnerability in the fingerd daemon as its primary attack vector [3]. Aleph1's seminal paper *Smashing the Stack for Fun and Profit* [4] publicised such attacks and prompted the widespread development of techniques to exploit such vulnerabilities. After many years of simply asking developers to write more secure code, operating system vendors and the designers of compilers decided to make changes to make exploitation more difficult for attackers. These anti exploitation measures now ship by default in most modern operating systems [5, 6, 7, 8] with some packages offering them as after market add-ons [9].

Memory corruption exploitation refers to the class of attacks that rely on ones ability to hijack the execution flow of a program by corrupting the applications memory space through a number of different possible attack vectors. The two most popular techniques of Stack and heap based exploitation are discussed below.

2.1 Stack Overflow Basics

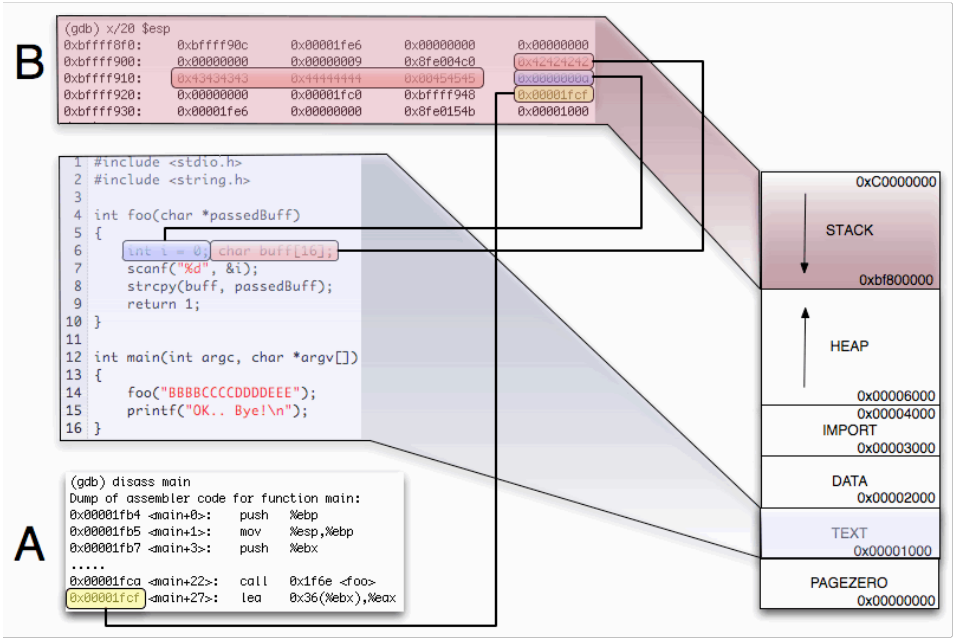

Figure 1 – Running program and its Memory Layout

The snippet of code in *Figure 1* is the canonical example of a typical stack based buffer overflow.

The disassembly of the main routine seen in the bottom left corner of the figure (labelled "A"), shows the address of the next instruction to be executed after the function foo() returns (0x00001fcf).

The top left portion of the diagram (labelled "B") shows the state of the Stack after the strcpy() function has run. We see how the characters "BBBBCCCCDDDDEEE" are copied on the stack which is growing downwards towards the other local variable (*int i*) and the saved return address (0x00001fcf).

As can be seen from the diagram, attempting to copy a buffer of greater than 15 chars length will result in strcpy() copying beyond the bounds of the buff buffer. Enough characters and the string can continue overwriting the integer *i*, the saved frame pointer, and eventually the saved return address.

Traditional stack overflow attacks aim at overwriting the saved return address on the stack. The plan would be to place executable code somewhere reachable in memory (possibly within the overflowed buffer). The address of this code is then used to overwrite the saved return address on the stack. When the function terminates, the overwritten address is popped off the stack, and execution returns to that location in memory.

2.2 Heap Overflow Basics

Heap overflows operate on a similar assumption to the traditional stack overflow, i.e. that the attacker has the ability to write beyond the bounds of a buffer. The major differentiator is that the heap does not hold a saved instruction pointer to overwrite, and is generally harder to tame. The overwriting of a saved function pointer on the heap [10] or overwriting of security sensitive values [11] are easy to understand and fairly commonly exploited but does not lend itself to a generic attack class.

A classic attack pattern relating to the heap however is known as "the arbitrary 4 byte overwrite". Heap allocations (and de-allocations) are managed by maintaining a doubly linked list. Each heap chunk that is allocated includes meta-data used for heap management. The information we care about for the purposes of exploitation is traditionally referred to as the flink and blink pointers (*ptr->next* and *ptr->previous*).

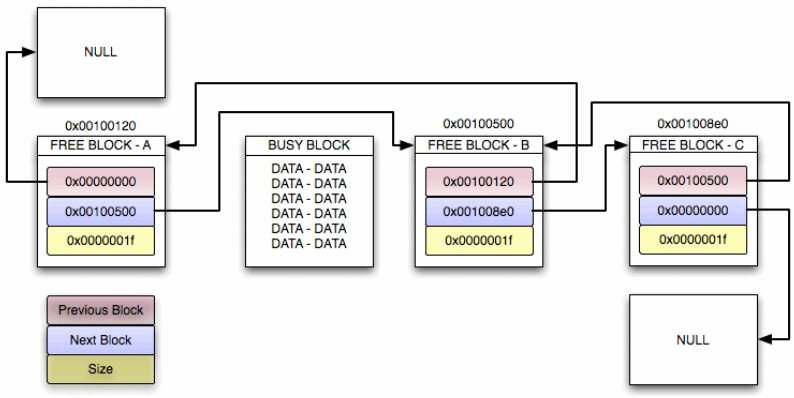

Figure 2 – The Free-List Linked List

Figure 2 is an example of the Free-List linked list, which maintains the chain of free heap memory on an OS X machine.

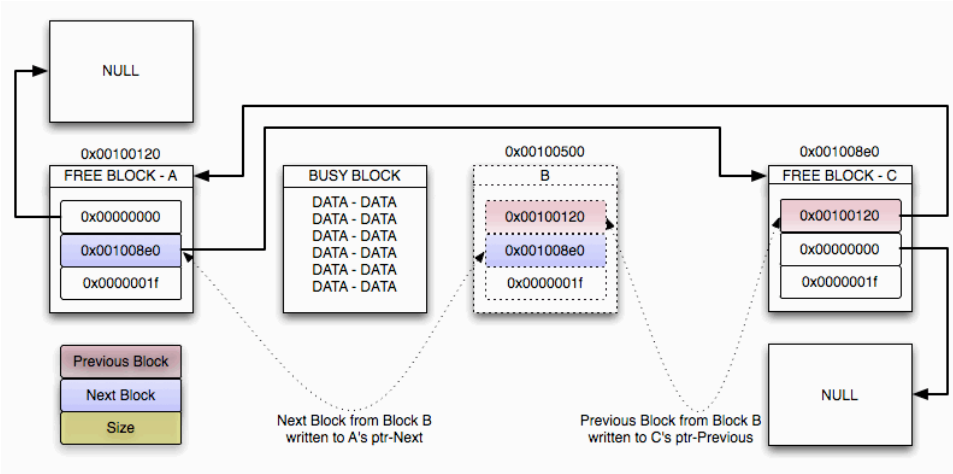

Figure 3 – The effects of an UNLINK operation

Figure 3 demonstrates what happens during a normal unlink operation, when a heap chunk is allocated. When FREE BLOCK-B is unlinked and removed from the free list, the unlink operation updates the *ptr->next* record of it's *ptr->previous* block (FREE BLOCK-A) with its own *ptr->next* value (0x001008e0) and updates the *ptr->previous* record of the block pointed to by its *ptr->next* block (FREE BLOCK-C) with it's own *ptr->previous* value (0x00100120).

Clearly an overflow in chunk-BUSY_BLOCK would allow an attacker to overwrite the meta-information of FREE BLOCK-B (0x00100500) including the *ptr->next* and *ptr->previous* pointers. Controlling these pointers means that we are able to write an arbitrary 4-byte value (taken from B's *ptr->next*) to an arbitrary location (pointed to by B's *ptr->previous*) in memory.

This attack vector is fairly well understood and explored in the Win32 and Linux worlds.

3 GENERIC DEFENCES (AND THEIR BYPASSES)

While specific defensive coding techniques are necessary to completely address vulnerabilities in code, several generic defenses have been introduced into most operating systems over the past few years. The most popular of these are discussed below.

3.1 Non Executable Stack

Although quickly worked around by researchers like Rafal Wojtczuk [12] and John McDonald [13], one of the first generic defences against memory corruption attacks was the introduction of the non-executable stack [14]. This protection (as implied by its name) aims to ensure that even if an attacker is able to redirect execution flow into his attacker controlled buffer (traditionally stored on the stack), the code would not execute, since the stack would be marked non-executable.

Today the Windows Family (XP, Vista, Win2k3) and Mac OS X (Leopard) operating systems make use of modern processor advances (NX bit) [15] to mark the STACK segment as non-executable. This can be tested fairly easily. To illustrate this, one can copy shellcode [16] to a locally declared buffer that is stored on the stack and make use of a function pointer to execute this code. The code can be seen below in *figure 4*.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  /* osx_ia32_bind -  LPORT=4444 Size=112 Encoder=PexFnstenvSub http://metasploit.com */
6  unsigned char scode[] =
7  "\x31\xc9\x83\xe9\xea\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x1f"
8  "\x77\x06\x47\xb0\x71\xa0\xc6\x89\x4b\x7d\x76\x8a\xaf\x12\xe8\x59";
9
10 int main(int argc, char *argv[])
11 {
12     void (*f)();
13     char stack_shellcode[113];
14     memcpy(stack_shellcode, scode, sizeof(scode));
15     f = (void (*)()) stack_shellcode;
16     f();
17 }
```

Figure 4 – Executing code from the Stack through a function pointer

Running the code in *Figure 4* causes a Segmentation Fault, and examining the program in a debugger yields the following error:

```
(gdb) run
Starting program: /Users/haroon/stuff/research/issa09/code/stack-exec
Reading symbols for shared libraries ++. done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0xbffff8bb
0xbffff8bb in ?? ()
```

Figure 5 – Kernel Failure when attempting to execute code form the stack

One can tell clearly that OS X has thrown an EXC_BAD_ACCESS error while trying to execute code at *0xbffff8bb* (an address on the stack).

3.2 Bypassing the Non-Executable Stack

Ret-2-libc (return to libc), another widely known attack pattern, evolved quickly to deal with non-executable stacks [17]. This pattern relies on the fact that even if the stack is marked as non-executable, library code is reachable within the processes memory space and is marked as executable. In its most common variant, the attacker will aim to overwrite saved EIP (Execution Instruction Pointer) with the location of the system command (which traditionally resides in libc, explaining its name), while preparing a fake stack frame containing the arguments to be passed to the system call. (In the canonical ret-2-libc attack, the attacker passes "/bin/sh" as the parameter to system, in order to launch a shell).

With enough creativity, multiple fake frames can be constructed with chained return-2-libc calls which can be used to devastating effect. In 2007 it was a ret-2-libc attack that did the heavy lifting of jail-breaking the iPhone after the initial libtiff vulnerability [18] was exploited [19].

3.3 Non Executable Heap

With the stack being off-limits as a destination for attacker supplied code, the next logical target is the process heap. Several innovative techniques sprung up to cater for this with one of the most impactful being the Heap Spraying technique described by Skylined [20]. Realising that calls to create or concatenate JavaScript strings result in these strings being created on the heap, Skylined made use of a simple loop construct to create multiple copies of his string (encoded shellcode) on the heap. He

then redirected execution to the heap where his shellcode would run. (The technique is called "spraying" since, in the absence of knowing the exact location of the shellcode on the heap, the attacker creates multiple copies of the shellcode on the heap (preceded by large NOP sleds) in order to increase the likelihood that a jump to the heap would result in code execution.)

Skylined's heap spraying attack targeted Internet Explorer but subsequent attacks have applied the same technique to Adobe Acrobat Reader [21] and SQL Server [22]. In 2006 Alexander Sotirov took this vector to a new level with a paper titled *Heap Feng Shui in JavaScript* [23], which refined the use of JavaScript to give surgical accuracy over the Heap for such attacks.

Starting with Windows XP-SP2, the heap is also marked as non-executable, to some degree mitigating this problem (unless a process specifically marks the page as executable instead.) While *vmmstat* on OS X reports the heap as non-executable, it appears as if the processors NX capability is not utilized to protect this area, allowing code to be executed on the heap. Sample code to validate this (which executes with no error) is shown in Figure *6*.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* osx_ia32_bind -  LPORT=4444 Size=112 Encoder=PexFnstenvSub http://metasploit.com */
unsigned char scode[] =
"\x31\xc9\x83\xe9\xea\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x1f"
"\x77\x06\x47\xb0\x71\xa0\xc6\x89\x4b\x7d\x76\x8a\xaf\x12\xe8\x59";

int main(int argc, char *argv[])
{
    void (*f)();
    char *heap_shellcode = malloc(sizeof(scode));
    memcpy(heap_shellcode, scode, sizeof(scode));
    f = (void (*)()) heap_shellcode;
    f();
}
```

Figure 6 – Code to execute shellcode from the heap

3.4 Safe un-linking of Heap Chunks

With XP-SP2 Microsoft introduced the concept of safe unlinking during memory management [24]. Before using the *flink* and *blink* pointers (discussed earlier), the heap allocator checks to ensure that *flink->blink* and *blink->flink* both point at the current block. This prevents an attacker from using the unlink operation to perform an arbitrary 4-byte overwrite. With XP-SP2, Microsoft also includes a single byte cookie in the heap metadata which is checked during unlink [25]. An incorrect cookie value indicates that heap corruption has taken place. Vista takes this protection further by encrypting important metadata (XORing the metadata with a random 32bit value) and decrypting it before use. At the time of writing no such protection exists within OS X Leopard.

3.5 Address Space Layout Randomization (ASLR)

All of the attacks discussed above rely on the attacker being able to predictably locate objects in memory. Without this ability, most remote execution attacks (resulting from memory corruption) can be mitigated down to process crashes. Following from the pioneering work on the PAX project [26], researcher Matt Miller released *WehnTrust* [27], which offered full address space layout randomization on the Windows platform, and Microsoft introduced ASLR with the release of Windows Vista. Early versions of the implementation suffered from flaws and extensive work was done by Ollie Whitehouse to examine the amount of randomness in the Vista ASLR implementation [28]. Today Windows XP, Vista and Server 2003 boast an ASLR implementation that poses a significant hurdle to an attacker who is un-aided by a supplementary bug that leaks memory layout information.

Apple's implementation of ASLR however leaves a lot to be desired and in its current incarnation has been dubbed by some researchers as Partial Library Randomization [29]. Leopard only randomizes the addresses of most libraries within the process memory space. This ignores the randomization of the stack, the heap, the image itself or even the address of some key libraries in a race where a single predictable location can result in the race being lost. In addition to this the current random locations are documented in the world readable file */var/db/dyld/dyld_shared_cache_i386.map* allowing system local attackers the full knowledge necessary to carry out an attack [29].

3.6 Compiler Level Protections

Crispin Cowan first introduced *Stack Guard* in 1998 [30]. A compile time protection, Stack Guard worked by placing a virtual canary on the stack in front of the saved return address. Any attempt to overwrite the return address would also result in altering the canary which is checked when the function returns. Microsoft independently created the /GS (Guard Stack) compiler flag to obtain the same results [31]. This protection has come under fire several times and Microsoft's '/GS' implementation has been through several iterations, until finally arriving at today's version which includes advanced heuristics at compile time to re-order variables on the stack. This means that in the example application shown in *Figure 1*, the compiler would have re-ordered the variables so that *buff* would not have been able to overflow the *i* integer.)

Although not as advanced in all respects as its /GS counterpart, Leopard ships with a version of GCC that supports stack protection through the *ProPolice* project and the *–fstack-protection* compile time option. While researchers like Whitehouse [28] and Maynor [32] have released tools to identify which binaries on a Vista machine are not compiled with /GS protection, the situation on Leopard is almost perfectly reversed, with the majority of applications on Leopard having not been compiled with this kind of protection enabled.

3.7 Caveat

We have ignored an entire attack pattern by failing to discuss the class of attacks directed against Structure Exception Handlers (SEH). The Windows OS makes use of a SEH routine that leaves the platform uniquely vulnerable to an attack pattern, which aims at replacing the structured exception handler with an address of our choosing, before causing an exception. Vista makes use of a new protection mechanism called SEHOP [33] to protect against such attacks. OS X like most Unix derivatives make use of signals as opposed to a default exception handler making Leopard not vulnerable to this class of attacks by default.

4 COMPARISONS

The findings so far are documented in *Table 1*, below.

Table 1 – Comparisons between Vista and Leopard

Attack Pattern	Windows Vista	OS X Leopard
Non-Executable Stack	YES	YES
Non-Executable Heap	YES	NO
Safe Heap Unlinking	YES	NO
A.S.L.R	FULL	Partial
Compiled with Stack Protection	Partial	Partial
S.E.H exploit protection	SEHOP	Not Applicable

It is fairly clear from *Table 1* that Apple lags significantly behind its Windows counterpart when it comes to generic anti-exploitation defences. What is unclear is why the lack of said defences has not led to some of the large scale attacks (like *Slammer,* or *Code-Red*), that have been witnessed against Windows machines in the past [34, 35].

A market share that is too small to attract real malevolence is an argument that is often made, but this does not stand up to the counterpoint of the literally hundreds of vulnerabilities reported weekly in obscure, relatively unused code-bases [36]. The argument that Apple simply makes less exploitable mistakes in their code is also untenable as researchers have had no difficulty exploiting OS X in public contests, when the need arose [37].

It is our belief that one of the explanations is that the client operating system OS X is being compared to the server operating systems in the Windows family. Both systems have different natural adversaries and so have different risk profiles. It can be postulated that OS X currently sits in an unusual niche, staying off the radar of server-attackers while below the threshold to make it an attractive target for attackers wishing to capture large volumes of desktop computers (for botnets or similar activities).

Apple would be well advised to make good use of their time in this niche to learn from the mistakes made by those before them, because as their market share steadily rises, they steadily inch closer to moving out of this protected space. They currently have a narrow window in which they can refine their defences. This would include a more robust ASLR implementation and can enforce the mandatory use of compile time stack protection to raise the bar on the requirements for a successful attack against the system.

5 CONCLUSION

We have demonstrated the basics of memory corruption exploits, and have examined how Microsoft Windows Vista and Apple's MacOS X Leopard combat these attacks in their default state. In this analysis OS X has been weighed and measured, and has come up wanting.

The next release of OS X, named Snow Leopard is currently in Beta and promises to improve the security posture of the system. It remains to be seen if the controls mentioned in this paper have been implemented or improved upon at all.

We hope that Apple is able to make the necessary improvements before it too is forced into altering its views on generic OS protection mechanisms through the media frenzy that follows public security breaches.

6 REFERENCES

[1] Rafal Lukawiecki. “Windows Vista Security”. http://download.microsoft.com/download/7/0/5/7058e678-6151-448e-a53b-43b83b5d309e/Windows%20Vista%20Security.ppt (2006)

[2] Jordan Hubbard. “OS X, From the Server Room to Your Pocket”. In proceedings of the 22nd Large Installation System Administration Conference. (12 November 2008)

[3] E. H. Spafford. “Crisis and aftermath”. In Communications of the ACM archive, Volume 32 , Issue 6 (June 1989) (Pages: 678 – 687)

[4] Aleph One. “Smashing the stack for fun and Profit”. Phrack Magazine 7, 49 (Fall 1997); http://www.phrack.com/issues.html?issue=49&id=14.

[5] “x86: Solaris Supports the no execute Bit”. Solaris10 Release Notes. http://docs.sun.com/app/docs/doc/817-0552/6mgbi4fgg?l=en&a=view&q=PROT_EXEC (2006)

[6] Theo de Raadt. “Exploit Mitigation Techniques (in OpenBSD).” http://www.openbsd.org/papers/auug04/index.html (2004).

[7] “Data Execution Prevention”, Microsoft Technet, http://technet.microsoft.com/en-us/library/cc738483.aspx (2009)

[8] Arjan van de Ven. “New Security Enhancements in Red Hat Enterprise Linux v.3, update 3”. (August 2004).

[9] “PaX”. http://pax.grsecurity.net/

[10] Solar Designer. “Bugtraq: Linux SuperProbe exploit”. http://seclists.org/bugtraq/1997/Mar/0011.html (05 Mar 1997).

[11] “Microsoft IIS HTR Chunked Encoding heap overflow allows arbitrary code “.Symantec security response center. (12 Jun 2002). http://www.symantec.com/security_response/vulnerability.jsp?bid=2033.

[12] Rafal Wojtczuk. “Defeating Solar Designer's Non-executable Stack Patch”. (30 January 1998). http://insecure.org/sploits/non-executable.stack.problems.html.

[13] John McDonald. “Bugtraq: Defeating Solaris/SPARC Non-Executable Stack Protection”. (03 Mar 1999). http://seclists.org/bugtraq/1999/Mar/0004.html.

[14] Solar Designer. “Non-Executable User Stack”. http://www.false.com/security/linux-stack/.

[15] http://en.wikipedia.org/wiki/NX_bit

[16] http://en.wikipedia.org/wiki/Shellcode

[17] Solar Designer. “lpr LIBC RETURN exploit”. (10 Aug 1997) http://insecure.org/sploits/linux.libc.return.lpr.sploit.html.

[18] “About the security content of iPhone v1.1.2 and iPod touch v1.1.2 Updates”. http://support.apple.com/kb/HT2170. (2007).

[19] Niacin, “iTouch/iPhone exploit source code released”, http://toc2rta.com/?q=node/30. (21 Oct 2007)

[20] Berend-Jan Wever (SkyLined). “Internet Exploiter 3: Technical details”. (01 Dec 2004). http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/details_msie_ani.html.php.

[21] Security Updates available for Adobe Reader and Acrobat versions 9 and earlier. (19 Feb 2009). http://www.adobe.com/support/security/advisories/apsa09-01.html

[22] Bernhard Mueller (SEC Consult Vulnerability Lab). “Microsoft SQL Server sp_replwritetovarbin limited memory overwrite vulnerability“. (09 Dec 2008). https://www.sec-consult.com/files/20081209_mssql-sp_replwritetovarbin_memwrite.txt.

[23] Sotirov, A. “Heap Feng Shui in JavaScript”. Blackhat Europe 2007. http://www.phreedom.org/research/heap-feng-shui/

[24] “A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2”. (26 Sep 2006). http://support.microsoft.com/kb/875352.

[25] Johnson, R. “Windows Vista Exploitation Countermeasure”. (29 Sep 2006). http://www.authorstream.com/presentation/Mentor-6833-rjohnson-Windows-Vista-Exploitation-Countermeasure-windows-vista-exploitation-countermeasures-ppt-powerpoint.

[26] “PaX”. http://pax.grsecurity.net/

[27] “WehnTrust, Host Intrusion Prevention System for Win2000, XP and Win2003”. http://www.codeplex.com/wehntrust

[28] Ollie Whitehouse, “An Analysis of Address Space Layout Randomization on Windows Vista”. (2007). http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Presentation/bh-dc-07-Whitehouse.pdf .

[29] Charlie Miller, Dino Dai Zovi, "The Mac Hackers Handbook", (2009). Wiley Publishing.

[30] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beat- tie, A. Grier, P. Wagle, Q. Zhang, “StackGuard: Automatic Adaptive Detection and

Prevention of Buffer-Overflow At- tacks". (1998). Proceedings of the 7th USENIX Security Conference.

[31] Brandon Bray, Visual Studio Team. "Compiler Security Checks In Depth". (Feb 2002). http://msdn.microsoft.com/en-us/library/aa290051.aspx.

[32] Maynor D. "Looking Glass". (10 Apr 2004). http://www.erratasec.com/lookingglass.html.

[33] "Windows Vista Service Pack 1 and Windows Server 2008 now include support for Structured Exception Handling Overwrite Protection (SEHOP)". (28 Jan 2009). http://support.microsoft.com/kb/956607/en-us.

[34] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. "Inside the Slammer Worm". IEEE Security & Privacy Magazine. (2003)

[35] David Moore, Colleen Shannon, Jeffery Brown. "Code-Red: a case study on the spread and victims of an Internet worm". Presented at the Internet Measurement Workshop (IMW). (2002)

[36] United States Computer Emergency Readiness Team (US-CERT). "Cyber Security Vulnerability Summaries per Week". http://www.us-cert.gov/cas/bulletins/.

[37] Matt Hines. "Mac Hacked Via Safari Browser in Pwn-2-Own Contest". http://securitywatch.eweek.com/apple/mac_hacked_via_safari_browser_in_pwn2own_contest.html. (20 Apr 2007)