NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

MEMORY CORRUPTION MITIGATIONS AND THEIR IMPLEMENTATION PROGRESS IN THIRD-PARTY WINDOWS APPLICATIONS

by

Serbulent Cevik

September 2012

Thesis Advisor:	Chris Eagle
Second Reader:	Dan C. Boger

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2012	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Memory Corruption Mitigations and Their Implementation Progress in Third-Party Windows Applications		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Serbulent Cevik			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number _____N/A_____.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE** A	

13. ABSTRACT (maximum 200 words)

It has been more than two decades since the first practical implementation of a memory corruption attack. Despite the fact that there has been much research done on efficiently protecting systems from this type of attack, memory corruption attacks still hold the lion's share among all exploitation techniques used against software systems. The Windows family of operating systems, as the most used operating system in the world, has suffered from memory corruption attacks more than any other system. Through the years, Microsoft has introduced various mechanisms for the detection and prevention of memory corruption attacks on Windows platforms. This thesis provides a timeline and detailed analysis of the memory protection mechanisms introduced in the Windows family of operating systems. Using these measures, Microsoft has diminished the number of successful exploitations of their software products, yet adoption of these measures by independent software vendors (ISVs) developing software for the Windows platform has not materialized as expected. The results of this thesis show that while most ISVs implement mitigations offered by Microsoft, few applications implement all these mitigations thoroughly.

14. SUBJECT TERMS Memory Corruption Attacks, GS, Data Execution Prevention, Address Space Layout Randomization, ISV, SafeSEH, Structured Exception Handling, Exploit Mitigations, Third-Party Applications, Buffer Overflow			**15. NUMBER OF PAGES** 85
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

MEMORY CORRUPTION MITIGATIONS AND THEIR IMPLEMENTATION PROGRESS IN THIRD-PARTY WINDOWS APPLICATIONS

Serbulent Cevik
Lieutenant, Turkish Army
B.S., Hacettepe University, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL
September 2012

Author: Serbulent Cevik

Approved by: Chris Eagle
Thesis Advisor

Dan C. Boger
Second Reader

Dan C. Boger
Chair, Department of Information Sciences

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

It has been more than two decades since the first practical implementation of a memory corruption attack. Despite the fact that there has been much research done on efficiently protecting systems from this type of attack, memory corruption attacks still hold the lion's share among all exploitation techniques used against software systems. The Windows family of operating systems, as the most used operating system in the world, has suffered from memory corruption attacks more than any other system. Through the years, Microsoft has introduced various mechanisms for the detection and prevention of memory corruption attacks on Windows platforms. This thesis provides a timeline and detailed analysis of the memory protection mechanisms introduced in the Windows family of operating systems. Using these measures, Microsoft has diminished the number of successful exploitations of their software products, yet adoption of these measures by independent software vendors (ISVs) developing software for the Windows platform has not materialized as expected. The results of this thesis show that while most ISVs implement mitigations offered by Microsoft, few applications implement all these mitigations thoroughly.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

API	APPLICATION PROGRAMMING INTERFACE
ASLR	ADDRESS SPACE LAYOUT RANDOMIZATION
CVE	COMMON VULNERABILITIES AND EXPOSURES
DEP	DATA EXECUTION PREVENTION
ISV	INDEPENDENT SOFTWARE VENDOR
MS	MICROSOFT
NX	NO EXECUTE
OS	OPERATING SYSTEM
PAE	PHYSICAL ADDRESS EXTENSION
PE	PORTABLE EXECUTABLE
PEB	PROCESS ENVIRONMENT BLOCK
PTE	PAGE TABLE ENTRY
ROP	RETURN ORIENTED PROGRAMMING
SDL	SECURE DEVELOPMENT LIFECYCLE
SEH	STRUCTURED EXCEPTION HANDLING
SEHOP	STRUCTURED EXCEPTION HANDLING OVERRIDE PROTECTION
SP	SERVICE PACK
TEB	THREAD ENVIRONMENT BLOCK
TWC	TRUSTWORTHY COMPUTING
VPTR	VIRTUAL POINTER
VS	VISUAL STUDIO
XOR	EXCLUSIVE OR

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Chris Eagle, and second reader, Dan Boger, for their patience and support in the completion of this thesis. I would like to thank Ollie Whitehouse for providing his analysis software and code. It made my life easier during the analysis process. I would also like to thank all the faculty of the Department of Information Sciences for the excellent education I experienced during my two years at NPS.

Lastly, I would like to thank my parents for always being there for me and supporting everything I do in life.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM

The year 2011 can be characterized as the "Year of the Hack." Myriad high-profile companies and government organizations fell victim to hacking attacks. Some of these attacks were quite sophisticated, while most were nothing new, in the sense that similar attacks had been seen before. The sheer volume of attacks got a lot of coverage in the news, which helped the public recognize the importance of computers in everyday life and, more importantly, the challenges in securing them.

Today, almost all organizations try to protect their systems with perimeter defenses, such as firewalls, intrusion-detection systems, and antivirus software. All of these solutions are intended to keep attackers out, but they may fail badly, as seen many times in the past. The problem lies in the fact that all of these solutions deal with symptoms rather than the source of the problem. Bad software is the main source of a vast majority of computer security problems. However, it is also known that there is no real way to write perfectly secure code. There will always be vulnerabilities somewhere in the program, and it is impractical to identify and remove them all.

Nowadays, a different approach to the "buggy" software problem is gaining strength in the software-development industry. Major software-development companies such as Microsoft and Google are trying to develop and integrate mitigation technologies that raise the bar against attackers exploiting their products. The sole purpose of these mitigations is to increase the cost and complexity of writing reliable exploits that can spread very quickly. These mitigations, in combination with secure software development practices, have proven to be successful for prominent software companies like Microsoft and Adobe.

Microsoft, as the leading operating system (OS) supplier, as well as other OS vendors, has incorporated most of these mitigation technologies in their products. Nevertheless, with the widespread use of the Internet, end-users have started to get

software from a variety of sources, including independent software vendors (ISV). Depending on an ISV's incentive, the security level of ISV software might be poor. A report by the security firm Secunia, which is limited to the Microsoft Windows platform, confirms this statement by showing that, in 2011, the majority of vulnerabilities (79%) were found in third-party applications (Secunia, 2012). Secunia's findings support the emerging consensus that attackers have shifted from finding and exploiting vulnerabilities in OSs to focusing on third-party applications. As a result, today we have a significant problem in ensuring the security of third-party applications.

B. RESEARCH OBJECTIVES

Most of the malicious attacks that compromise the security of end systems exploit low-level programming errors, which are called vulnerabilities, in target programs. Attackers exploit vulnerabilities such as stack overflow and heap overflow to launch memory corruption attacks that enable them to control the target program, and subsequently the target end system. In the last decade, a wide range of defense mechanisms against memory corruption attacks has been proposed, and some were adopted by the industry. The purpose of this thesis is to present:

- An overview of the most common memory corruption vulnerabilities found in software and their exploitation mechanisms
- Principal defense mechanisms proposed by either academia or industry against memory corruption attacks
- A detailed look at the defenses implemented by Microsoft
- An analysis of popular third-party applications for their implementation of defenses offered by Microsoft

C. ORGANIZATION OF THE THESIS

This thesis is organized as follows. Chapter II familiarizes the reader with the fundamentals of memory corruption vulnerabilities, attacks, and defenses. Chapter III provides a detailed look at the defenses, or mitigations, offered by Microsoft in their OSs

and development suites. Chapter IV presents analysis of popular third-party applications, for adoption of these defenses and discussion of the findings. Finally, Chapter V summarizes the basic insights achieved, the significance of the study, and recommendations.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. PRELIMINARY

1. Processes

A process is an instance of a program running concurrently with other programs on a computer. These programs are mostly stored on disk in binary form, which is called an executable image, and these images include instructions and data. The steps taken from compiling a program from source code to loading its image from disk to memory (for execution) can be seen in Figure 1.

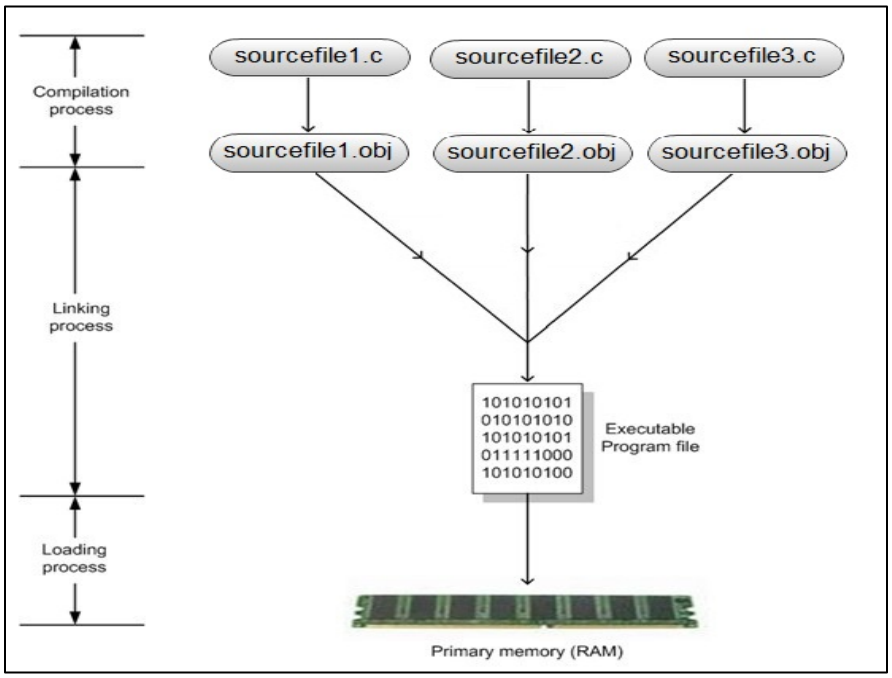

Figure 1. Compile–link–load stages for running a program (a process)

Basically, a compiler transforms source code written in a high-level language into machine language that can be executed by processors.[1] The output of the compilation is called an object file. The object files generated by a compiler are combined by a linker to produce a single, executable file. In the Microsoft Windows family of OSs, the standard

[1] Assembler is considered as part of the compiler.

executable file format is called portable executable (PE) format. When a file is executed, the binary file in PE format is read by the Windows loader into memory and the OS passes control to the process.

At any time, there are multiple processes running on a computer. Modern multitasking operating systems schedule processor time among these processes. In order to switch between different processes, the OS keeps track of the last instruction executed by each process. This is called program counter or instruction pointer (IP). When a process is stopped, its IP is saved. When this process regains processor next time, its IP is used to resume execution where it left off.

2. Memory Layout of a Process

Any process running on a computer has its own address space, which includes that process' instructions and data. From process' point of view, it has access to entire physical memory, but in reality, the OS shares this limited resource among processes via virtual memory management. Figure 2 shows a typical layout of process' segments in memory.

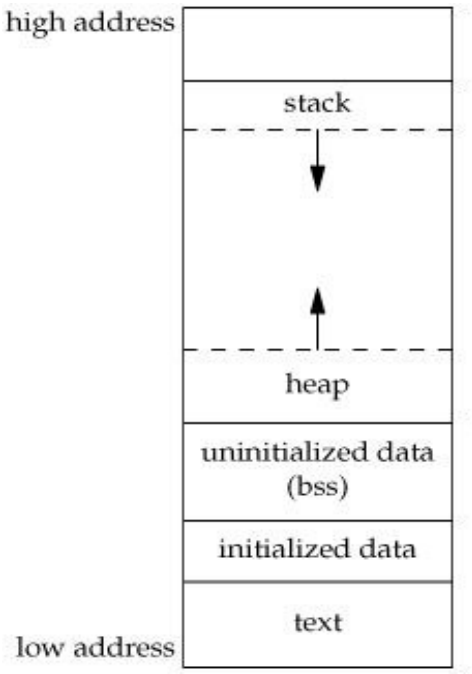

Figure 2. Typical memory layout of a process

The address space of a process is divided into logical segments with different access permissions. These segments are defined in executable files. The text segment holds instructions of a process. It is defined as read-only so as not to allow the process to modify its instructions at runtime. Data and bss segments both hold static variables of the process. To save space on disk, two separate segments are defined. The data segment is used for initialized data and the bss segment is for uninitialized data. Only the size of the bss segment is stored in the binary image.

A stack is a “last in, first out” (LIFO) data structure that allows two basic operations: push and pop. The push operation puts a new value on the top of the stack, and the pop operation removes the most recent value from the stack. The stack segment holds automatic variables, control-flow information, exception records, and other temporary information required for nesting of functions. The stack segment grows

downwards, towards lower memory addresses. More information about the stack will be given in the section, “Functions and the Stack.”

The heap segment is where dynamic variables are stored. These variables can be allocated and released at runtime. Unlike stack segments, heap segments grow upwards, towards higher memory addresses.

Segment Name	Permissions	Description
text	Read-Execute	Contains instructions of the process
data	Read-Write	Contains initialized static variables of the process
bss	Read-Write	Contains uninitialized static variables of the process
stack	Read-Write	Contains automatic variables
heap	Read-Write	Contains dynamically allocated variables

Table 1. Memory segments in a process

3. Registers

Registers are small amounts of memory directly connected to the processor. They provide the fastest way to access data. Machine instructions use registers as variables, encoded as part of the machine instructions, which are defined in the instruction set of the processor. For the purposes of this thesis, registers can be categorized into four groups:

- General purpose
- Segment
- Instruction pointer
- Other

General purpose registers are used for holding variables for mathematical or any other operation. They can hold data, address, offset, index variable, etc. This group of registers includes EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP. On a 32-bit processor, these registers are 32 bits in size, but 16-bit and 8-bit access also exists for backward compatibility. Table 2 shows all access methods for general purpose registers and their envisioned usage.

32 bit	16 bit	8 bit (high)	8 bit (low)	Description

EAX	AX	AH	AL	Accumulator
EBX	BX	BH	BL	Base index (for use with arrays)
ECX	CX	CH	CL	Counter
EDX	DX	DH	DL	Data/general
ESI	SI			Source index for string operations
EDI	DI			Destination index for string operations
EBP	BP			Base pointer for the current stack frame
ESP	SP			Stack pointer

Table 2. General Purpose Registers

Among these general purpose registers, ESP and EBP deserve a closer look. ESP points to the top of the stack where the next stack operation will take place. Its value is changed every time an instruction directly or indirectly manipulates the stack. By contrast, EBP keeps the base address of a function's stack frame and is heavily used for accessing automatic variables defined in the scope of the function. The next section covers functions and their inner workings in more detail.

Segment registers, such as CS, DS, SS, ES, FS, GS, are used to keep track of segments and allow backward compatibility for real-mode operation. These registers are all 16-bit in size. The code segment (CS) must be set to an executable segment, and the stack segment (SS) must be set to a writable data segment. The others are all used for data segments.

The instruction pointer register, extended instruction pointer (EIP), contains the address of the machine instruction that will be executed next. If this register is successfully overwritten, the flow of execution can be controlled. It is incremented automatically after fetching a new instruction, but this sequential nature can be altered by machine instructions like subroutine calls, jumps, branches, and returns.

There are many other registers with different functionalities such as control registers, test registers, debug registers, flags register, etc. For example, the extended flags (EFLAGS) register comprises 32 bits showing the current state of the processor.

General purpose registers and the instruction pointer register are the main registers discussed in this thesis.

4. Functions and the Stack

Functions are the building blocks of any programming language. They allow the programmer to break a complex task into smaller, manageable subtasks. Programs are composed of functions, each of which has a specific task. The linear control flow of a process can be altered by calling functions. A function can be called from anywhere in the program and, after performing, returns to the next instruction following the call instruction.

To return back to the point where a function is called, the processor needs to remember the address that follows the call instruction. Function calls also can be cascaded, which requires the processor to remember the return addresses. The stack is used to keep these addresses. When a process calls a function, it automatically stores the value of its EIP on the stack, and restores it after execution finishes in the function. This enables the process to restore its original control flow.

Functions require memory space to store temporary variables. These local variables are also stored on the stack. At the start of each function, the required amount of space for local variables is allocated on the stack. If the functions accept arguments, they are also passed to the function using the stack. Each function has a reserved area consisting of its arguments, local variables, and all information needed to restore the stack to its original form when the function finishes execution. All this information is called the stack frame. Figure 3 shows a function’s stack frame.

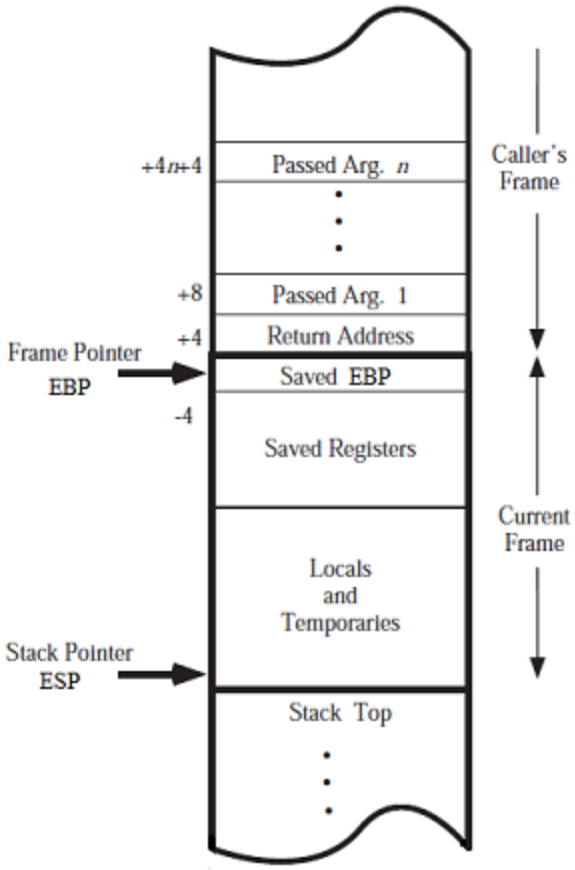

Figure 3. Example stack frame

Local variables and arguments should be accessible throughout a function's lifetime. ESP cannot be used for this purpose because its value changes every time the program uses the stack; therefore, the base pointer (EBP) is used. When a function starts, it first saves the previous value of EBP to the stack, and then copies the value of ESP to EBP. After this point, variables and arguments are accessed relatively, using EBP. The instruction block for setting up EBP and allocating space for local variables on the stack is called the function prologue. Likewise, the instruction block used to release local variables, restore the stack pointer, and return from the function is called the function epilogue. The basic code block in the prologue and epilogue of a function is displayed in Figure 4.

```
1                               ; prologue
2   push    ebp                 ; save base pointer
3   mov     ebp, esp            ; copy stack pointer into base pointer
4   sub     esp, 20             ; allocate space for local variables
5
6
7                               ; epilogue
8   mov     esp, ebp            ; release local variables
9   pop     ebp                 ; restore base pointer
10  ret                         ; return from function
```

Figure 4. Prologue and epilogue of a function

B. MEMORY CORRUPTION VULNERABILITIES

1. Buffer Overflows

Buffer overflow vulnerabilities are the most common programming errors exploited by malicious parties in the last decade (Mitre, 2012). It was first mentioned as early as 1972 (Anderson, 1972), but the first significant exploitation of a buffer overflow vulnerability dates back to 1988, when the Morris worm took advantage of a buffer overflow in the finger[2] service (Spafford, 1989). Despite research and protections implemented by software developers over the years, buffer overflows are holding ground as the most widely exploited vulnerability.

A buffer is a region of memory with a specific size, used for keeping data. If the data written to a buffer exceeds its size, it overflows to adjacent memory locations, where there may be other data, metadata, or program control-flow data. When this metadata or control-flow data is overwritten, it may be possible to redirect the program's execution flow. Buffer overflow vulnerabilities are called after the memory segment where these buffers are stored.

[2] Finger service is a service that dispenses information about the set of users logged into a UNIX-based computer system.

a. Stack Overflows

Stack overflow is the first type of buffer overflow vulnerability that caught public attention, and is exploited widely in the wild. Typically, stack overflows occur as a result of improper bounds-checking by string-handling functions in the C and C++ languages. Figure 5 shows a very basic piece of code that is vulnerable to stack overflow. There are various ways to exploit stack overflow vulnerabilities (Pincus & Baker, 2004).

```
1  #include <stdio.h>
2
3  main()
4  {
5      char buffer[128];
6      gets(buffer);        /* No bounds checking */
7  }
8
9
```

Figure 5. Basic stack overflow vulnerability

(1) Stack Smashing. This technique involves the overwriting of the return address (saved EIP) on the stack with a value supplied by the malicious user. Typically, the malicious user also supplies a piece of code, called shellcode[3] or payload, as a part of the buffer. When the function returns, control flow is redirected to the address intended by the malicious user, and the shellcode is executed. Stack smashing was first introduced by Aleph One in his classic article, "Smashing the Stack for Fun and Profit" (Aleph One, 1996).

(2) Off-by-One Overflow. An off-by-one overflow happens when a stack buffer overflows just one byte into the saved frame pointer (EBP) on the stack (Klog, 1999). Mostly this happens as a result of an overlooked NULL byte appended as the last character in a string operation, misunderstanding of how array indexing works, or misuse of operators (Dowd, McDonald, & Schuh, 2006). For example, if the saved frame pointer on the stack holds the value 0xABAD1DEA and there is an off-by-one overflow

[3] This piece of code is called shell code because it is often used to spawn a shell (access to the computer through a command-line interface).

vulnerability in the current function, when this function restores the value of the previous frame pointer in its epilogue, an attacker can overwrite the last byte with NULL, making the value of restored frame pointer 0xABAD1D00. This modified address lies in the range of the overflowed buffer because the stack grows towards lower addresses. When the caller function wants to return, it uses the modified frame pointer to access its stack frame, so it ends up restoring the instruction pointer from the overflowed buffer, which includes attacker-supplied values. There are two restrictions for this technique. First, this approach is valid if the processor architecture is little endian.[4] Second, the buffer should be next to the saved frame pointer on the stack.

(3) Return-to-library. Return-to-library is a useful exploitation technique where the vulnerable program has some form of memory protection that does not allow a segment of memory to be both writable and executable. A common approach in this kind of situation is to overwrite the saved instruction pointer with the address of a function that resides in a library that is already loaded into the program's address space. There are caveats to this approach. The address of the function should be known beforehand, and the arguments needed by the called function should also be supplied on the stack. Generalizing this approach enables the chaining of useful functions, called one after another, allowing the implementation of more complex tasks. This exploitation technique is mostly referred as "return to libc" because it was first introduced in Linux systems, and functions in the libc, the standard C library loaded into nearly every Linux program, were used in the early implementations of this technique (Solar Designer, 1997).

(4) Pointer Subterfuge. Pointer subterfuge is the general name given to exploitation techniques that involve overwriting of a pointer stored on the stack, other than the saved instruction pointer. Function pointers, data pointers, exception handler pointers, and virtual pointer (VPTR) are some of these pointers that have been exploited through years. These techniques were developed by attackers in response to

[4] A little endian machine stores the least significant byte first.

integrity-checking mechanisms (which will be mentioned later) introduced against stack smashing attacks.

Function pointers are used in C-like languages to simplify code by providing a simple access to a function and executing it based on runtime values. If a function pointer is stored as a local variable following a buffer on the stack, an attacker can trigger an overflow to modify the value of this pointer. Subsequently, any call to this function pointer will transfer control flow to the place of the attacker's choice. This technique is first described by Matt Conover in his paper on heap overflows (Conover, 1999).

Another variation of function-pointer overwrite techniques is to overwrite the VPTR field stored in an object in memory (Rix, 2000). C++ stores a table of virtual function pointers and accesses this table using VPTR. These pointers are used for implementing dynamic binding at runtime, which enables polymorphism. An attacker can change the value of VPTR using a buffer overflow and make it point to any other place in memory. This allows the attacker to take control of execution when the next virtual function is invoked using this table. VPTR overwrite can also be possible on the heap if the object is created at runtime.

The Windows family of OSs uses structured exception handling (SEH) mechanism for handling exceptions that occur as a result of any event outside the normal execution of a program. Exception handlers are defined in a program for resolving these exceptions and avoiding unexpected termination. Also, these handlers are chained in a linked list structure, and when an exception occurs, they are called in orderly fashion until one of them resolves the problem. The pointers for this mechanism are stored on the stack, so they are susceptible to buffer overflows. This technique is also called exception-handler hijacking (Litchfield, 2003).

Finally, attackers also make use of data pointers for exploiting buffer overflows. If an attacker overflows into a data pointer, and this pointer is used as a target in a subsequent assignment, the attacker can modify any memory location arbitrarily (Bulba & Kil3r, 2000). The data-pointer-overwriting technique is mostly used in combination with other techniques discussed for exploitation of buffer overflows.

b. Heap Overflows

The heap is the other memory region where buffer overflow vulnerabilities are exploited. Heaps are used by processes to store their dynamic variables and data with varying sizes at runtime. Unlike the stack, heaps grow towards higher memory addresses. In Windows, each process has a default heap, but there can be more than one heap present in any process. The heap manager, which is part of the OS, provides functions, such as allocating and releasing memory, for managing heaps. Memory allocations from the heap can be done using `HeapAlloc` in Windows API, `malloc` in C, and `new` in C++. Similarly, memory can be released using the `HeapFree`, `free`, and `delete` function calls, respectively.

The heap manager divides heap segments into contiguous chunks that include both user and metadata. This metadata includes size, location, and other management information. Figure 6 shows how used and free heap chunks look. The heap manager keeps track of the allocated and free chunks with the help of this metadata stored next to user data. When a chunk of memory is allocated, a pointer to that chunk is returned to the caller. Size fields in allocated chunks allow the heap manager to walk forward and backward through chunks in the heap segment. When the process releases an allocated chunk of memory, this freed chunk is appended to a doubly linked list of free chunks with the same size, and pointers in the metadata of the chunk are updated to reflect this change. The heap manager uses the pointers in the chunks and doubly linked lists for tracking free blocks of memory with the same size. As more allocations and releases occur through the process' lifetime, these pointers and lists are updated accordingly. Lastly, the heap manager does not allow two contiguous free chunks in memory to avoid fragmentation. If two free chunks come next to each other, they are coalesced into a larger free chunk. This consolidation is called unlinking.

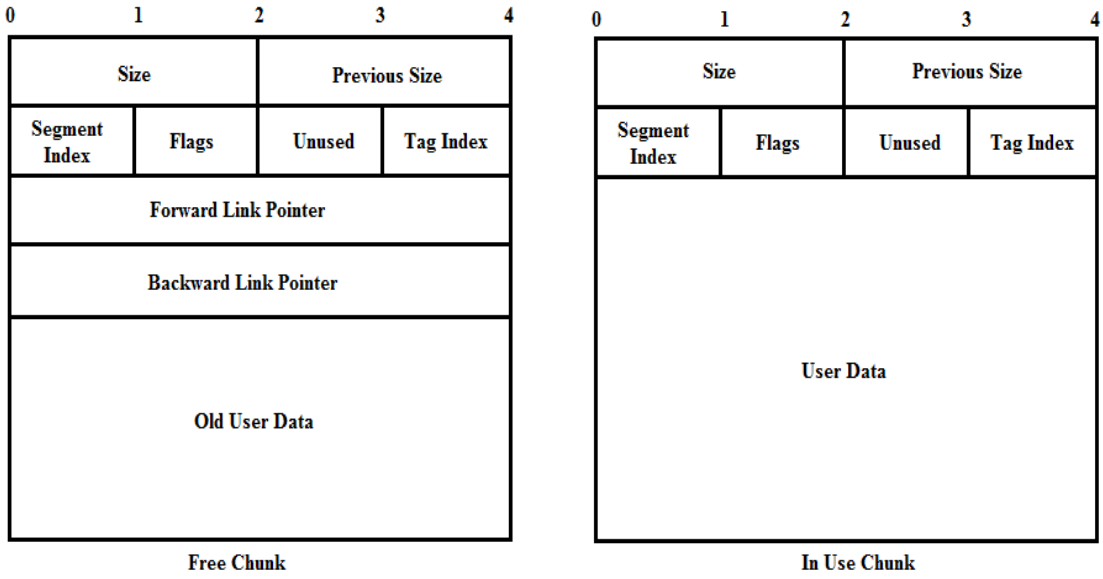

Figure 6. Free and in-use heap chunks

The first implementations of heap overflow attacks targeted application data and function pointers kept on the heap (Conover, 1999). These attacks were very similar to their stack counterparts. Later, heap overflow attacks targeting heap metadata and the specifics of dynamic memory allocators were introduced (Solar Designer, 2000). This section will examine the details of the latter one.

By overflowing a buffer on the heap, an attacker can overwrite the metadata of the adjacent chunk. Heap-based buffer overflow vulnerabilities can be exploited when a chunk of memory is freed, and heap manager tries to update the doubly linked lists of free chunks. At this point, with the help of forced unlink operation, heap-based buffer overflows allow the attacker to write an arbitrary value anywhere in memory. Potential targets can be function pointers, pointers to exception handlers, etc. Figure 7 illustrates a basic piece of code that is vulnerable to heap-based buffer overflow.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char *buff1, *buff2;
6
7      buff1 = malloc(40);
8      buff2 = malloc(40);
9
10     gets(buff1);    /* No bounds checking */
11     free(buff1);
12 }
13
```

Figure 7. Basic heap overflow vulnerability

2. Format String Vulnerabilities

Format functions, the `printf` family of functions, are used in C like languages for generating formatted output. These functions support a variable number of arguments, so they depend on the format string, which is also an argument, in deciding how many arguments they should expect. The format string is exactly copied to output unless a "%" character is encountered. This character, followed by a format specifier, allows the format functions to generate output according to the specifier. When a format specifier requires an argument, it is found on the stack. Some of the format specifiers used to exploit format string vulnerabilities are as follows:

- "%s" prints a string to output, so format function expects a pointer to a char on the stack
- "%d" prints the argument on the stack in decimal integer format
- "%x" prints the argument on the stack in hexadecimal format
- "%n" writes the number of bytes printed to output so far to a memory location pointed by a pointer, which is supplied as an argument on the stack
- "%p" expects a pointer as an argument on the stack, and prints it in hexadecimal format

A format string vulnerability occurs when format functions are used improperly by programmers, for example, (`printf(buffer)` instead of `printf("%s",`

`buffer),` where buffer is a user-supplied string). This allows an attacker to control what is read from the stack as the argument and write any value to any memory location (Scut, 2001), so an attacker can exploit format string vulnerabilities to control execution flow by overwriting a function pointer or any other interesting memory location.

3. Integer Errors

Integer errors (Blexim, 2002) are not directly exploitable vulnerabilities, but they can be used in combination with the other techniques to achieve exploitation of the program. There are two kinds of integer errors: integer overflows and integer sign errors.

An integer overflow occurs when an integer variable is assigned a value that is larger than its upper limit. According to standard ISO/IEC 9899:1999, when this happens, the value is set to modulo of the variable with MAXINT+1 (i.e., 65536 + 2, will store the value 1 in an unsigned 16-bit short integer). This new value of the variable may become too small, which may lead to other vulnerabilities discussed above. In a scenario where a program allocates a buffer on the heap using the integer overflow variable, this may lead to heap-based buffer overflow vulnerability, because the length of the buffer becomes too small for holding the user-supplied input. An attacker can take control of the program by first overflowing an integer variable and then exploiting the newly created vulnerable situation.

In C-like languages, integers are interpreted as signed or unsigned. Unless explicitly stated otherwise, all integers are signed. Implicit casting is applied to a signed integer variable when an unsigned variable is expected, or vice versa. An integer sign error occurs because of this implicit casting operation and may lead to exploitation of a program. For example, an attacker may pass a signed integer with a negative value as an argument to a function that will use it in a memory copy operation. This negative value can pass any maximum size tests before it is used, but it will be implicitly type casted to unsigned integer for the memory copy operation (e.g., `memcpy` in C is used to copy the

contents of memory from one location to another, and it expects an unsigned value as its size parameter). This situation may cause buffer overflow vulnerability on the heap or stack, and an attacker can use it to take control of the program.

4. Dangling Pointers

Pointers are special data types holding memory addresses as their value, which are used for accessing the value kept in that memory location. Dangling pointers is the general name given to pointers that do not point to a valid object in memory. This happens as a result of not automatically updating the pointers when the memory location they point is no longer valid. There are two basic types of dangling pointer vulnerabilities, based on the use of already freed memory location.

The first type of vulnerabilities occurs because of the reuse of a dangling pointer that points to a freed memory location. The attacker can validly change the contents of the memory location after it is freed in the first place, so that later, when the dangling pointer is dereferenced (the pointer is used for accessing the contents of the memory location it points to), the attacker can access what he put in that memory location. For example, the dangling pointer may be a pointer to a function in a freed C++ object. If the attacker manages to allocate the same memory location and put shellcode to this very location, when the dangling pointer is dereferenced, it will jump to the attacker's shellcode.

The other dangling pointer vulnerability occurs when a released memory location is released a second time. This type of dangling pointer vulnerability is mostly called a double-free vulnerability in the literature (Dobrovitski, 2003). For example, if a program releases an object stored on the heap, a freed chunk is added to the free chunks list by the heap manager. If this memory location is erroneously freed again, the heap manager will add the chunk to the same list again, thus causing the chunk to point to itself in the list. In this situation, an attacker can force the program to allocate this double-freed chunk and overwrite pointer fields of the chunk with attacker-supplied data, because it is allocated. Then, a second allocation request with the same size will cause the heap manager to

return the very same chunk. This time, the unlink operation will use the pointer fields, assuming the chunk is free, but these fields will contain what was put in by attacker, allowing the attacker to overwrite any memory location with any value.

There are two other variations in dangling pointers: uninitialized reads and invalid frees. These errors can also cause corruption of memory and allow an attacker to take control of execution flow. Uninitialized-read vulnerabilities occur when a program reads from newly allocated objects that contain data from previous allocations. On the other hand, the invalid-free type of vulnerability is the result of a program trying to release a memory location that is not allocated in the first place. In this thesis, these two types of vulnerabilities are not covered in detail.

C. DEFENSES AGAINST MEMORY CORRUPTION VULNERABILITIES

In the previous section, the basics of various memory corruption vulnerabilities and their exploitation techniques were discussed. During the past decade, numerous defense mechanisms against memory corruption attacks have been proposed by both academia and the security industry. Some of these mechanisms focus on removing the vulnerabilities in the first place, by using safe languages, safe libraries, static analyzers, etc., while others try to prevent the successful exploitation of the related vulnerability, such as a non-executable stack and stack canaries, or make the attacker's job harder, as seen in address space or instruction-set randomization. This section will provide a list of notable defense mechanisms organized into categories. An overview of each mechanism and its limitations will be discussed briefly.

1. Safe Languages

Most of the memory corruption vulnerabilities are found in the code written in unsafe languages like C and C++. There are lots of security pitfalls inherent in these two languages, which can make the programs written in these languages vulnerable to exploitation. While the use of safe languages like C# and Java has increased considerably

since their introduction, C and C++ remain popular for various reasons, such as performance (Tiobe, 2012). Also, there is much legacy software written in these languages and still deployed in organizations.

Safe languages eliminate memory corruption vulnerabilities by disallowing pointer arithmetic, unsafe memory operations, memory deallocation, etc. Programmers are not given explicit control over the management of dynamic memory (i.e., the garbage collection mechanism only deallocates memory when there are no references pointing to it, thus preventing dangling pointers from referring to deallocated memory). Safe languages also employ strong typing, which restricts the operations that attempt to disregard data types. Type safety can be provided at compile time (static typing) or run time (dynamic typing) by safe languages.

Instead of switching to a new language, programmers can take advantage of safe-language features by using a C dialect. These dialects prevent memory corruption vulnerabilities while retaining C's syntax and semantics. Cyclone (Jim, et al., 2002) is one of these dialects, introduced as a safe alternative to the C language with the caveat of performance overhead. Cyclone implements garbage collection instead of manual memory management.

2. Static Analysis

Programmers can prevent memory corruption vulnerabilities before the deployment of the software. Static code-analysis tools examine the source code of a program, either lexically or semantically, for finding potential vulnerabilities. Since these tools lack runtime information, their coverage is limited. A more important problem with these tools is the high false-positive rate.

Lexical analyzers do not try to understand the semantics of the code. They parse the source code and try to find suspicious constructs that may lead to vulnerability. This simple analysis is only capable of discovering basic vulnerabilities. Flawfinder

(http://www.dwheeler.com/flawfinder/) is a powerful lexical analysis tool that supports both C and C++. It keeps a signature database of well-known problematic constructs in C and C++ languages and checks the source code against these signatures.

The second type of static analyzers checks the source code based on the semantics of the program. They consist of two modules: a front-end parser and a back-end analyzer. The parser creates a model from the source code and the analyzer employs formal methods or heuristics to find vulnerabilities in the model. The coverage of semantic analyzers is generally better than their lexical counterpart, and they can find more complex and obscure problems. BOON, BLAST, and PolySpace for C/C++ are some well-known examples for semantics-based static analyzers.

3. Compiler Extensions and Libraries

Early solutions to memory corruption attacks focused on preventing specific exploitation methods. They were not general solutions to the bigger problem. Most of these defense mechanisms came in the form of compiler extensions or runtime libraries and were limited to only providing defense against a specific exploitation technique only.

StackGuard (Cowan, et al., 1998) is a good example of this approach, as it only protects programs from stack smashing attacks. It inserts a canary[5] value between ESP and EIP on the call stack and verifies the canary's integrity before the function returns. If this value is changed, StackGuard assumes the saved EIP on the stack is overwritten by an overflow, and terminates the execution of the program immediately. Another similar integrity-based concept was offered by Stack Shield (Vendicator, 2000). In this mechanism, the saved EIP is copied into a table in the data segment (in the function prologue) and it is copied back to the stack at function exit (in the function epilogue). These two mechanisms prevent only buffer overflow attacks that target the saved EIP, so there are various ways to bypass these defenses (Bulba & Kil3r, 2000).

[5] The value on the stack is named after canaries in coal mines which are more sensitive to toxic gases and used as biological warning system by miners.

As indicated before, there may be other valuable targets on the stack, such as function pointers and data pointers, which can be overwritten in a memory corruption attack. ProPolice (Etoh & Yoda, 2001), an enhancement to StackGuard's concept, overcomes this limitation by introducing the reordering of local variables stored on the stack. With ProPolice, array variables are stored at the highest addresses on the stack frame to prevent an overflow into other local variables. It also protects the integrity of a function's arguments by making a copy and relocating them with local variables. Microsoft uses the defense techniques discussed so far in its Visual Studio C++ compiler by providing a compiler option (/GS), which will be discussed comprehensively in the next chapter.

Libsafe and Libverify are two library-based solutions suggested against buffer overflows on the stack. These libraries are loaded dynamically at runtime, so they do not require recompilation of the program unless statically linked. Libsafe intercepts every call to vulnerable library functions, such as `strcpy`, and replaces these calls with safer alternatives, which guarantees that a buffer cannot exceed stack frame boundaries. Libsafe also provides protection against format string attacks by rejecting any attempts to overwrite saved EIP using a "%n" specifier. By contrast, Libverify uses a technique similar to that of Stack Shield, but the canary values are stored on the heap.

A more general approach to memory corruption attacks was proposed by PointGuard (Cowan, Beattie, Johansen, & Wagle, 2003). Attackers are mostly interested in overwriting pointers because they give the most advantage for controlling execution flow. PointGuard's protection consists of encrypting the pointer values before they are stored in memory and decrypting them just before they are copied to registers. The encryption scheme is simple linear XOR operation, so it is possible for an attacker to brute force the overwriting of the least significant byte and to succeed in redirecting execution flow to an intended address (Alexander, 2005). Microsoft provides APIs in its latest OSs to encode and decode pointer values at the discretion of the programmer.

Buffer overflows are not limited to the stack, so there are various defenses offered against heap-based buffer overflows. ContraPolice (Krennmair, 2003), the heap

counterpart of ProPolice, implements canary-based defenses on the heap, but again, this kind of solution protects only the integrity of inline heap metadata. There may be other valuable targets for attackers stored in the data portion, which is not protected by these mechanisms (such as function pointers). Heap canaries are also used by the Windows family of OSs.

FormatGuard (Cowan, Barringer, Beattie, & Kroah-Hartman, 2001) is a library extension that provides safer alternatives to the `printf` family of functions, to protect programs from format string attacks. First, it counts the number of arguments passed to the functions and then compares this number with the number of "%" directives in the format string. If the latter is more than the number of provided arguments, it is assumed a format string attack attempt, and the execution of the program is aborted.

4. Randomization-Based Defenses

The success of diversity in retaining robustness in biological systems inspired security researchers to use the same concept to prevent IT monoculture[6] and create more secure computer ecosystems (Forrest, Somayaji, & Ackley, 1997). Diversification in software systems is needed to decrease the number of common vulnerabilities, which in turn decreases the probability of large-scale exploitation of these vulnerabilities. Diversity can be introduced into software systems by building software from functionally same but structurally different components. In the context of memory corruption attacks, randomization is widely used to introduce diversity into programs in order to make exploitation of vulnerabilities harder for the attacker. Randomization can be applied to memory layout, instruction set, or data space.

Most of the exploitation techniques used in the wild depend on overwriting return addresses or function pointers with a known address that mostly points to the shellcode supplied by attackers. Address space layout randomization (ASLR) prevents these attacks by making prediction of these hard-coded addresses more difficult. ASLR assigns

[6] IT monoculture is a result of having large number of software instances running in a computational ecosystem with little or no diversity.

different base addresses to key memory regions in the process' address space, such as executable image, libraries, stack, heap, etc., for each execution of the program. ASLR also protects programs from return-to-library attacks because the base addresses of libraries are randomized as well. ASLR was first implemented as a part of PaX project – a security patch for the Linux kernel (PaX, 2001). Microsoft has supported ASLR since 2007. The effectiveness of ASLR depends on the entropy of randomized base addresses. Brute-force guessing attacks were shown to be successful against low entropy offered by ASLR defenses on 32-bit systems (Shacham, et al., 2004). ASLR is also susceptible to information-leakage attacks, such as attacks exploiting format string vulnerabilities. Heap spraying is the most recent technique for evading ASLR defenses. Spraying the heap with copious amounts of large heap objects containing malicious code would increase the attacker's chances of redirecting execution into one of these objects and evading ASLR.

Instruction set randomization (ISR) is another mechanism that introduces diversity into programs. ISR prevents attacks that try to inject malicious code into a program's address space. The idea behind ISR is to provide every process its own instruction set. An attacker needs to know in advance which instruction set is used by a target process, in order to inject malicious code to the process' address space. Process-specific instruction sets are mostly generated using encryption. Instructions are kept encrypted while in memory and decrypted right before they are executed. XOR encryption, which is not a strong encryption scheme, is mostly used for performance reasons in different ISR implementations. The main problem in these implementations is poor performance, because they are mostly emulator-based solutions, but this can be surpassed by hardware support. Since every process has its own instruction set and encrypted code, shared libraries become another problem point for ISR (Barrantes, Ackley, Palmer, Stefanovic, & Zovi, 2003).

A third type of randomization technique, inspired by PointGuard, randomizes data stored in program memory (Bhatkar & Sekar, 2008). Unlike PointGuard, which only protects pointer data, data space randomization (DSR) randomizes all types of data in memory. Data in memory is protected by encrypting (XOR) with random masks. Both

control and non-control data attacks can be prevented with DSR, but this requires masking and unmasking operations for each memory access, which degrades performance.

5. Non-Executable Memory

The attackers' ultimate goal in most of the memory corruption attacks has always been to execute malicious code injected into target-system memory as part of these attacks. To prevent attacks involving code injection, various defense mechanisms have been proposed to separate writable and executable portions of memory. These defenses block any attempt to execute code in writable memory, thus rendering code-injection attacks ineffective. Non-executable memory protections can be divided into two groups: software (emulation) based and hardware based.

Processors architectures like Harvard and Burroughs (tagged architecture) separate program and data portions of memory at the hardware level, so the processor can execute instructions from program memory only and can only write data in data memory. Memory corruption attacks that inject code into the data portion of memory and execute it are not applicable in these architectures. On the other hand, most commodity computer systems today use von Neumann architecture, which stores programs and data in the same memory. Thus, malicious code injected into data memory (e.g., the stack or heap) can be executed in computers supporting this architecture. Figure 8 shows the difference between Harvard and von Neumann architectures. The NX (No eXecute) bit is introduced in von Neumann architecture-based processors to prevent execution of code from the data portion of memory. An OS running on a processor that supports this feature can mark any portion of memory as writable or executable (but not both) at page level.

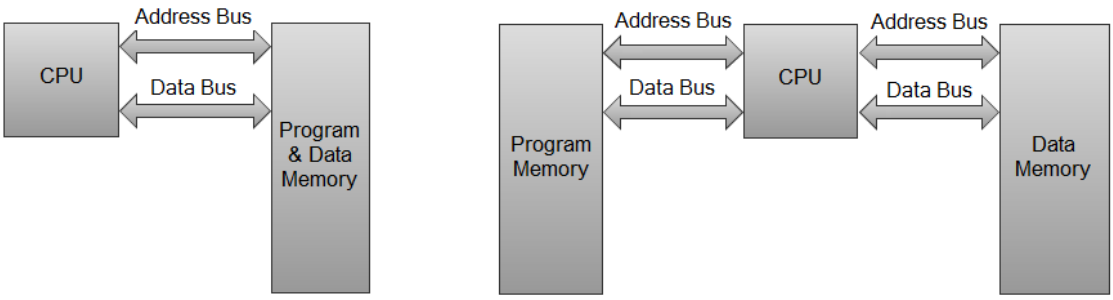

Figure 8. Von Neumann vs. Harvard architectures

Prior to NX bit support from processor manufacturers, a few software-based solutions were proposed to emulate this feature. ExecShield (van de Ven, 2004) and PaX's PAGEEXEC (PaX, 2001) are the two widely accepted solutions that provide page-level, non-executable memory protection for processors lacking NX-bit support. They both make use of segment limits feature in these processors to separate virtual memory into executable and non-executable halves.

There are two problems in using non-executable memory defenses. First, some programs (e.g., JIT compilers, video encoders) depend on generating code at runtime and executing it. They use the stack or heap for storing code generated on the fly, and execute code from these memory portions, which are designed for data. This is exactly the opposite of what non-executable memory defenses try to accomplish. The second problem is return-to-library attacks, where the attacker does not inject code, and instead executes library functions in a chained fashion to achieve his goal. Non-executable memory defenses fall short in attacks involving code-reuse techniques, such as return to library. Return Oriented Programming (ROP) (Shacham H. , 2007) was introduced as an extension to return-to-library attacks and showed that it is possible to evade non-executable memory defenses by chaining sequences of instruction blocks ending with "RET" instructions. These instruction sequences, which perform useful actions for the purposes of the attacker (i.e., pop a value stored on the stack, store it in a register, and return), are called gadgets.

6. Control-Flow Integrity

In a typical memory corruption attack, the attacker aims to alter the control flow of a target program by, for example, overwriting a return address or a function pointer. Depending on the exploitation technique used, the destination of this illegal control transfer may be the injected shellcode (code-injection techniques) or code that is already present in program's memory (return-to-library techniques). In both of these attacks, the attacker manages to change the expected control flow of the target program.

Control-flow integrity (CFI) (Abadi, Budiu, Erlingsson, & Ligatti, 2005) is a defense mechanism that protects programs from this type of illegal control transfers by enforcing a predetermined control-flow graph (CFG). CFG specifies all of the control-flow transfers that are allowed by the program and can be defined statically (source code or binary analysis) or dynamically (execution profiling). CFI implementation enforces CFG on a program by inserting runtime checks (binary rewriting) for instrumentation. CFI has a limitation, in that it requires binaries to be compiled with debug information, which may not be the case most of the time.

Program shepherding (Kiriansky, Bruening, & Amarasinghe, 2002) is another defense mechanism that also provides control-flow integrity for a program. It employs dynamic instrumentation of a program for enforcing a broad range of security policies at runtime. An extra piece of software (DynamoRIO - http://www.dynamorio.org/) is used for implementing instrumentation which results in poor performance.

CFI defenses have proven successful against attacks targeting control data with an overhead factor. However, they are ineffective against non-control data attacks (Chen, et al., 2005) that do not violate predetermined CFGs.

7. Taint Tracking

Non-control data attacks are similar to control-data attacks in that they use the same techniques, such as buffer overflows or format string attacks. They differ in the target of exploitation. As seen before, control-data attacks exploit various vulnerabilities to overwrite a return address, function pointer, or any other control data. Non-control data

attacks exploit similar vulnerabilities, but target security-critical data (e.g., user identification data, the user's privilege level, the server configuration string) in program memory. Non-control data attacks do not try to divert the control flow of the target program.

Taint tracking is a powerful technique for detecting both control and non-control data attacks. It is a form of data flow analysis that marks data introduced from untrusted sources (e.g., command-line arguments, network sockets) as tainted and keeps track of propagation of this tainted data during program execution. If tainted data is used as a source operand in any instruction, the destination also becomes tainted. Using this simple principle, it is possible to differentiate any address or data that is tainted by input from untrusted sources. If tainted data is used as an address to access data or code, it is regarded as a potential memory corruption attack, and execution is stopped.

Taint tracking can be implemented using various techniques, such as binary rewriting (Newsome & Song, 2005), compiler-based extensions (Xu, Bhatkar, & Sekar, 2006), and architectural support (Suh, Lee, Zhang, & Devadas, 2004). However, most of these techniques have high false-positive rates and overhead and otherwise require hardware support.

8. Sandboxing

Sandboxing is a security enforcement mechanism for creating limited execution environments, which are used for running untrusted programs or components of programs. It is not a defense mechanism especially designed against memory corruption attacks; however, with appropriate security policies, the damage from these attacks can be contained using sandboxing. The concept of sandboxing is first introduced in the form of software fault isolation (Wahbe, Lucco, Anderson, & Graham, 1993). Sandboxing can be implemented at various levels, such as the language (Java and C#), process (protected mode in Internet Explorer), and kernel (OS level virtualization).

Extensibility requirements of many commercial, off-the-shelf (COTS) programs (e.g., browsers, text editors, media players) pose a great risk to the security of these

programs. Untrusted third-party code introduced as extensions drastically increases the attack surface. To contain this problem, user-level sandboxing presents a valuable defense-in-depth mechanism to alleviate the impacts of attacks involving these extensions.

D. SUMMARY

Despite the large amount of research done on protection techniques, memory corruption attacks persist as one of the biggest problems in information security. The ideal solution to the problem is writing secure code that is free of vulnerabilities. However, there is a large legacy code base written in unsafe languages such as C and C++, and developers still opt in favor of these languages when it comes to performance. Among the proposed solutions, those that do not require any significant modification to programs or special hardware support have seen wide adoption in the software industry. Canary-based schemes, non-executable memory, randomized address space, etc., are a few examples of defense mechanisms that are easy to implement and do not induce significant performance penalty.

Since the introduction of Windows XP Service Pack (SP) 2 and Windows Server 2003, Microsoft has started to implement some of these defense mechanisms in their products. In the next chapter, the defense mechanisms offered by Microsoft to ISVs to protect applications against memory corruption attacks will be observed in detail.

THIS PAGE INTENTIONALLY LEFT BLANK

III. MEMORY CORRUPTION DEFENSES IN WINDOWS

> In the past, we've made our software and services more compelling for users by adding new features and functionality, and by making our platform richly extensible. We've done a terrific job at that, but all those great features won't matter unless customers trust our software. So now, when we face a choice between adding features and resolving security issues, we need to choose security. (Gates, 2002)

Ten years ago, Bill Gates, then chief software architect of Microsoft, sent an email to all employees announcing the start of the Trustworthy Computing (TwC) initiative. In the above citation from that letter, it is clearly seen that Microsoft has chosen increasing the security and reliability of their products over adding new features. This was an imperative decision for Microsoft because, security-wise, the company had experienced an awful year in 2001. CodeRed and Nimda, the two worms that affected Microsoft's Internet Information Server (IIS) product, had a massive impact on the newly developing Internet. These two worms took down a large number of public and private organization's networks, and some organizations voluntarily took down their networks as a precaution against these threats.

After Gates' TwC memo, Microsoft started "security pushes" to improve existing products by eliminating bugs, reducing the attack surface, training developers, etc. Lessons learned from these security pushes resulted in initiation of the "Secure Development Lifecycle" (SDL), which constitutes the core of TwC. Microsoft describes the SDL as "a security assurance process that focuses on software development and introduces security and privacy throughout all phases of the development process" (Lipner, et al., 2011). The SDL has been an integral part of the company's software development and maintenance practices since its introduction in 2004. It has helped Microsoft to lessen the number and severity of vulnerabilities in their products. Microsoft has also shared its SDL practices with the software industry since the release of version 3.2 in 2008. As seen in Figure 9, the SDL practices are gathered under the seven phases of the traditional software development process.

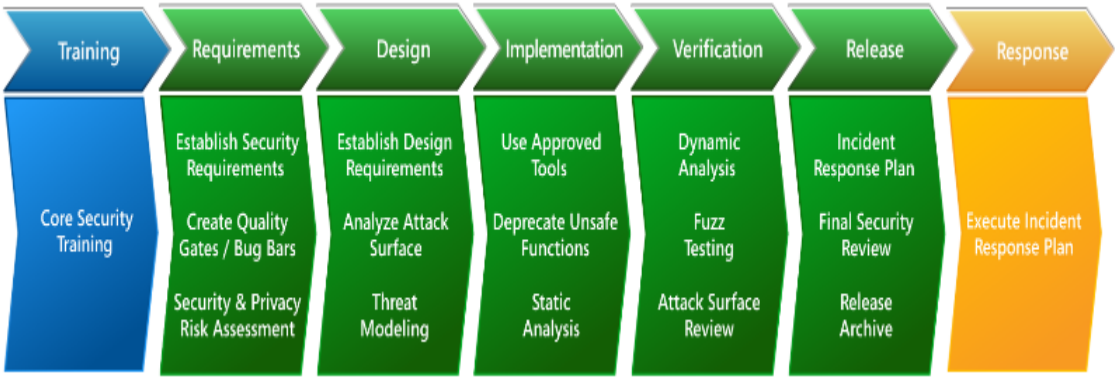

Figure 9. SDL process and practices (From: Lipner, et al., 2011)

The SDL is a success for Microsoft, but it is not a silver bullet for developing software free of vulnerabilities. Attackers are still able to find new vulnerabilities in Microsoft products or they can exploit old vulnerabilities on systems that are not properly patched. To solve this problem, Microsoft follows a defense-in-depth strategy to protect their customers against various threats. Security features such as firewalls, access control lists, patch management, configuration hardening, and exploit mitigations are implemented at different levels to protect users from malicious attackers.

This chapter examines the various exploit mitigation technologies against memory corruption attacks introduced by Microsoft through the years. These are:

- Stack-based buffer overflow detection (GS)
- Heap protections
- Pointer encoding
- Safe exception handling (SafeSEH and SEHOP)
- Data execution prevention (DEP)
- Address space layout randomization (ASLR)

These mitigations are intended to make the exploitation of vulnerabilities impossible, or at least hard and costly enough to deter attackers. The increased time and cost of creating exploits gains software developers enough time to address the vulnerability and deploy a patch; likewise, it protects users from attackers during this

time window. These mitigations have been widely used in Windows and other Microsoft products for a while now, but in order to prevent exploitation of third-party applications, these mitigations should also be adopted by ISVs. Some of these mitigation technologies are supported by Microsoft's Visual Studio (VS) software development suite, and others are integrated into the Windows family of OSs.

A. STACK-BASED BUFFER OVERFLOW DETECTION

Stack-based buffer overflow detection was first introduced to the C/C++ compiler in VS2002 and has been improved in subsequent releases of Microsoft's software development suite. It employs techniques like those used by StackGuard and ProPolice, which are discussed in the previous chapter. This detection capability is also known as GS, because Microsoft stores the original canary or cookie value in the data segment, which is accessed using GS segment register. It can be enabled by passing `/GS` option to the compiler and disabled by `/GS-` option (it is on by default since VS2003).

The GS option inserts a canary value, a four-byte, pseudo-random value, between the saved return address (old EIP) and the local variables on the stack. This random value is calculated by applying logical XOR operation to the master canary value (created once at load-time and stored in the data segment) and the saved return address. If a buffer overflow occurs on the stack, it will also overwrite the canary value, because buffer overflow attacks overwrite contiguous ranges of memory. Since the value of the canary is assumed to be unpredictable, there will be a mismatch when it is checked against the master canary value before the function returns, and the program will be terminated. As seen in Figure 10, extra pieces of code are added to the prologue and epilogue of the protected functions for creating the canary on the stack and checking its value.

While the GS option prevents successful exploitation of a subset of stack-based buffer overflows, it incurs a performance overhead because of the additional runtime checks, code, and storage. For performance reasons, GS protection is applied selectively

to functions that are susceptible to buffer overflow attacks. The decision as to whether a function is susceptible is based on heuristics applied by the compiler. A function is only protected if it possesses:

- An array that is larger than 4 bytes, has more than two elements, and has an element type that is not a pointer type
- A data structure whose size is more than 8 bytes and contains no pointers
- A buffer allocated using the `_alloca` function
- Any class or structure that contains a GS buffer

In 2007, this selective policy failed to protect a file compiled with GS option. After the successful exploitation of the animated cursor (.ani) vulnerability (CVE-2007-0038), Microsoft released VS2005 SP1 which included a new pragma[7], `#pragma strict_gs_check`, which can be used to force the compiler to apply GS protections more comprehensively.

```
1
2  ; prologue with GS option
3  push  ebp
4  mov   ebp, esp
5  sub   esp, 24h  ; allocate space for local variables (extra 4 bytes for the canary)
6  mov   eax, dword ptr [___security_cookie]   ; copy the original canary from data segment
7  xor   eax, dword ptr [ebp+4]    ; XOR it with the saved return address (old EIP)
8  mov   dword ptr [ebp-4], eax    ; store the canary
9
10
11
12 ; epilogue with GS option
13 mov   ecx, dword ptr [ebp-4]    ; get the canary from the stack
14 xor   ecx, dword ptr [ebp+4]    ; XOR it with the saved return address (old EIP)
15 jmp   __security_check_cookie   ; check the canary
16 mov esp, ebp
17 pop ebp
18 ret
19
```

Figure 10. Prologue and epilogue of a GS protected function

[7] Pragmas are compiler directives used for telling the compiler to implement specific features.

Another protection offered by GS option (since VS2005) is the reordering of local variables on the stack to prevent buffers from overflowing into other local variables. As discussed in the previous chapter, there are various memory corruption attacks that exploit critical variables on the stack, such as function pointers, to redirect execution flow. With variable reordering, buffers are moved to higher memory addresses than non-buffer, local variables on the stack, so such attacks are rendered ineffective. Additionally, parameter shadowing is used for protecting each GS-protected function's arguments. Potentially vulnerable arguments, such as pointers and buffers, are copied to lower memory addresses than local buffers, and instead of originals, these shadowed copies are used whenever needed. Figure 11 depicts the different layouts of the same function on the stack in the absence and presence of the GS option.

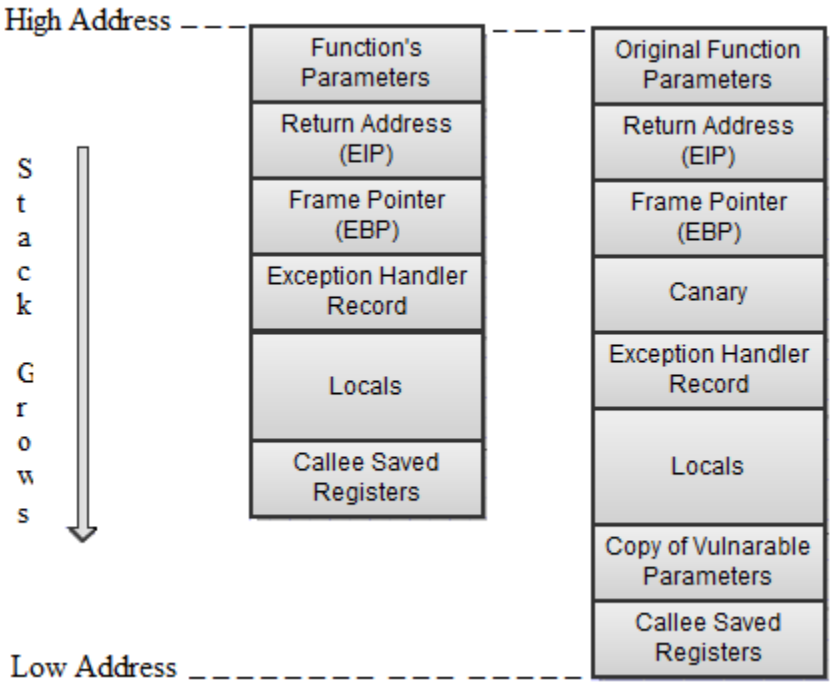

Figure 11. Stack frame layouts of the same function, with and without GS protection

B. SAFE EXCEPTION HANDLING

Exceptions are erroneous events, such as division by zero, that occur during a program's execution and require special handling to recover from these events, or in some cases to execute housekeeping code and terminate the program. Software developers using C-like languages put code that is susceptible to create exceptions under blocks known as "try," and code written for handling these exceptions is put under "catch" blocks. Figure 12 shows a piece of code that uses the exception handling mechanism.

```
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5     char *buffer;
6     try {
7        buffer = new char[256];
8        if( buffer == 0 )
9           throw "Error in memory allocation!";
10    }
11    catch( char * str ) {
12       cout << "Exception occured: " << str << '\n';
13    }
14 }
15
```

Figure 12. Sample code using exception handling mechanism

In Windows, two modes of exception handling, Structured Exception Handling (SEH) and Vectored Exception Handling (VEH), are provided to software developers. VEH is an extension to the SEH mechanism and differs from SEH in that VEH neither uses `try/catch` statements (it uses API calls instead) nor is tied to the stack frame of a specific function, which is the case for SEH. Exploit writers have mostly abused the SEH mechanism (as in the well-known Code Red worm), because of the poor design choice of keeping structures required by SEH on the stack next to local variables.

Microsoft implements SEH at the thread[8] level. Each thread has a pointer to a list of SEH records (`_EXCEPTION_REGISTRATION_RECORD`) that are created at runtime on the stack. These structures contain two pointers: one for accessing the exception handling code (`catch` block) and the other one for accessing the next SEH record on the list. This singly linked list of exception handler records ends with a default exception handler. When an exception occurs, the OS traverses this list until it finds a proper handler or reaches the end. The SEH records are valuable targets for attackers because they contain pointers to handlers, which are called whenever an exception occurs. As shown in Figure 11, SEH records are not protected by the canary mechanism.

Microsoft introduced a new mechanism known as SafeSEH (beginning with VS2003) to mitigate attacks that hijack SEH records on the stack to alter execution flow. SafeSEH can be enabled by passing `/SafeSEH` option to the linker. The linker adds a static list of legitimate exception handlers as metadata to the binary file. More precisely, the list of legitimate pointers is stored in an optional header known as "Load Configuration" which is a part of the PE file. At runtime, when an exception occurs, the exception dispatcher routine, which is part of the OS, checks this list of pointers to verify the authenticity of the exception handler found on the stack.

SafeSEH is an effective mitigation against memory corruption attacks that target SEH records on the stack, but it requires re-linking of already deployed binaries to protect them. Starting with Windows Vista SP1 and Server 2008 OSs, Microsoft introduced structured exception handling overwrite protection (SEHOP) which does not require re-linking and protects applications against SEH corruption. It works by walking down the list of exception handlers using the pointers in SEH records till it reaches the final exception handler (`FinalExceptionHandler` function resides in `ntdll.dll` and is initially added to the list as the last record). If an attacker overwrites the pointer to

[8] A thread is a lightweight process which is the basic unit of processing that can be scheduled by an operating system.

the next SEH record, the validation walk will not reach the final exception handler, which signals the corruption of SEH records list. In Windows 7, SEHOP can be enabled per application basis at the registry level.

C. HEAP PROTECTIONS

Microsoft's Windows family of OSs, as described in the previous chapter, uses free-list-based allocators to manage the heap. To keep track of the chunks allocated and freed on the heap, the Windows heap manager stores heap management metadata in-band next to the data. Also, like many free-list based allocators, it performs coalescing, the merging of free chunks into a larger chunk, to avoid memory fragmentation. Attackers traditionally relied on these structures and behaviors of the Windows heap manager for exploiting heap-based, memory corruption vulnerabilities.

Starting with Windows XP SP2 and Windows Server 2003 SP1, Microsoft introduced safe unlinking, which replaced the former vulnerable unlink operation. Before safe unlinking, attackers were able to write four bytes of their choice to an address of their choice (write4 primitive) by exploiting the coalescing operation of the heap manager. To prevent this technique, the new version of unlink operation, safe unlink, includes a validity check of pointers in the metadata of freed chunks to verify that they are consistent with each other. Also, with the release of safe unlinking, Microsoft started to add canary values to the metadata portion of the heap chunks. Like the stack canary, the heap canary provides an integrity-based detection mechanism against memory corruptions on the heap.

Microsoft gradually added additional protections against heap corruptions in Windows Vista, Windows 7, and Windows Server 2008. The heap canary's role was extended to cover more fields in the metadata section of the chunks. Important fields in metadata are now encrypted by XORing with a random value, and whenever accessed, these fields are decrypted and verified for integrity. Look-aside lists[9], which are widely exploited by attackers, were replaced by safer structures. A new API call

[9] A look-aside list is a list of fixed-sized blocks used for fast memory allocation.

(`HeapSetInformation`) was added to enable software developers to dynamically instruct termination of the program whenever a heap-based memory corruption is detected. The allocation algorithms used by the heap manager were revised to prevent attackers from predicting allocation patterns and abusing them with brute-force attacks.

D. POINTER ENCODING

Pointers, especially function pointers, have become the most valuable target for attackers after Microsoft implemented stack-based memory protection mechanisms such as stack canary, variable reordering and safe exception handling. Overwritten function pointers, either on the stack or on the heap, allow attackers to redirect the execution flow of the program when these functions are called. As discussed in the previous chapter, protections, such as PointGuard (Cowan, Beattie, Johansen, & Wagle, 2003) store pointers encrypted while they reside in memory and decrypt them when they are accessed. This kind of protection thwarts attackers from overwriting and exploiting function pointers.

With Windows XP SP2 and Windows Server 2003 SP1, Microsoft introduced a pair of API calls: `EncodePointer/DecodePointer` and `EncodeSystemPointer/DecodeSystemPointer.` These APIs use the XOR operation for encoding and decoding, similar to PointGuard's implementation. The first pair uses a random number that is generated per application basis, while the second pair uses a global random number generated per system basis at each boot.

E. DATA EXECUTION PREVENTION

For many years, attackers have relied on injecting malicious code in memory sections that are supposed to contain only program data (stack, heap, and data segments), and later forcing the program to redirect its control flow into this injected code. The introduction of data execution prevention (DEP) technology has rendered this basic exploitation technique ineffective in Windows family of OSs. DEP enables programs to

declare memory sections as either executable or writable (but not both at the same time) at page level. The implementation of DEP requires both software and hardware components.

The hardware support to mark memory pages exclusively as executable or writable was first introduced by Advanced Micro Devices (AMD) under the name No eXecute (NX) and followed by Intel's introduction of eXecute Disable (XD) technology. Both of these main processor manufacturers implement these supportive technologies by allocating a bit field (63rd bit) in the page table entry (PTE)[10] for marking a related page as executable (the bit is set to 0) or not executable (if the bit is set to 1). Since 32-bit, x86 processor architectures only have 32 bits for page table entries, the physical address extension (PAE) feature, which allows the processor to access more than 4GB of physical memory, should be enabled.

DEP has been available in Windows OSs running on processors that support the NX bit starting from Windows XP SP2 and Windows Server 2003 SP1. Microsoft maintains DEP settings in two levels: program and system. Developers can enable DEP for programs by setting the NXCOMPAT flag in the `DllCharacteristics` field of PE header. VS linker supports an option, `/NXCOMPAT,` that sets this flag. Microsoft also supports the `SetProcessDEPPolicy` API call for programmatically setting DEP policy at runtime. System-level DEP policy can be enforced in four different modes:

- Opt-In: Programs need to explicitly enable DEP (the default configuration for client OSs, such as Windows 7)
- Opt-Out: DEP is enabled for all programs which do not disable it explicitly (the default configuration for server OSs, such as Windows Server 2008)
- Always On: DEP is enabled for all programs without exception
- Always Off: DEP is disabled for all programs without exception

[10] PTEs are data structures used by OS's memory manager to keep track of memory pages.

The reason behind this distinction is the fact that many applications, either legacy or not, rely on generating code at runtime; storing it on heap, stack, or data segments; and executing it. With the help of these configuration options, compatibility issues are resolved by the setting proper mode of operation for different programs. Lastly, there is no “software DEP,” which appears in error messages of some Windows OSs running on processors with no NX-bit support, because it is nothing but the SafeSEH mitigation technology discussed before.

F. ADDRESS SPACE LAYOUT RANDOMIZATION

Most of the exploitation techniques seen in the wild replace return addresses or function pointers on the stack with the address of injected malicious code or library functions (as in return-to-library attacks). What is common in all of these attacks is that attackers assume that the memory layout of the target program is fixed, so they can hardcode the address where they want to redirect the execution flow. Address Space Layout Randomization (ASLR) is a feature that breaks this assumption by changing the layout of the programs in each boot of the system, as seen in Figure 13. The increased entropy as a result of ASLR makes writing reliable exploits that will work in every target harder. Especially, ASLR is an effective mitigation technology against fast spreading worms, which have taken advantage of software monoculture for many years.

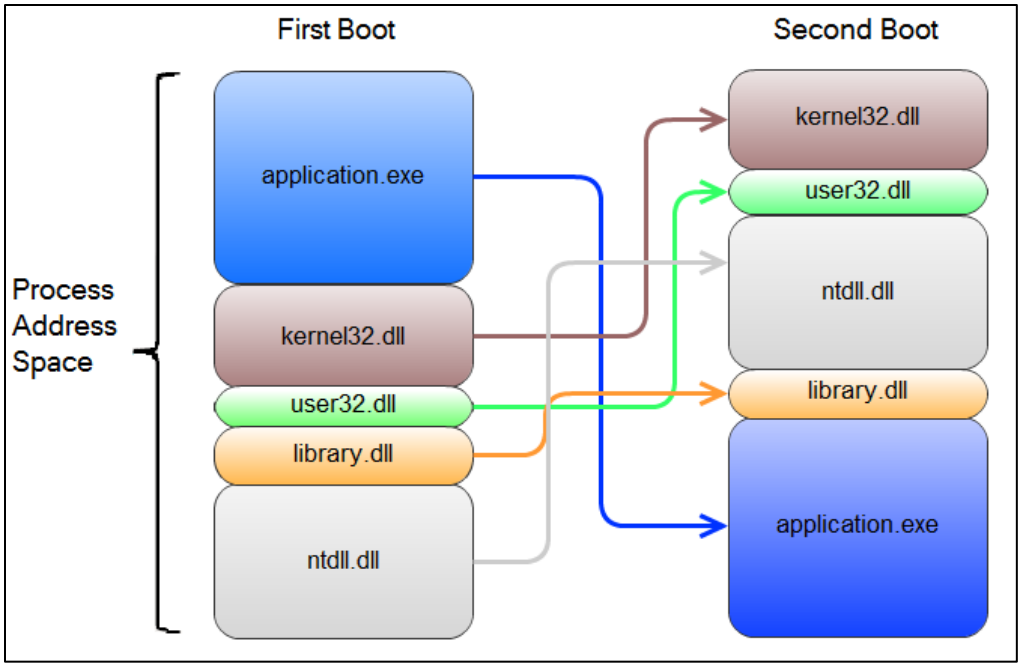

Figure 13. Visualization of ASLR

Although the ASLR concept has been known for a long time (PaX, 2001), Microsoft was able to implement it first in 2007 with the release of Windows Vista OS. However, Microsoft used randomization before the release of ASLR (since Windows XP SP2) to prevent attackers from exploiting internal data structures such as the process environment block (PEB) and thread environment block (TEB)[11]. With ASLR, the base addresses of executable images, stacks, and heaps are also randomized, besides PEB and TEB. There is a subtle difference between the randomization of these elements: the base addresses for PEB, TEB, stack, and heap are randomized per each execution, while the base addresses for other segments in the executable and dynamic libraries are randomized per each boot. Also, the amount of entropy obtained for these program structures differs in Windows' ASLR implementation (Whitehouse, 2007). Table 3 shows the amount of effective entropy provided by ASLR for different program structures.

[11] PEB and TEB are internal data structures used by Windows OS kernel for keeping track of processes and threads running on the system.

	Entropy Provided (in number of bits)
Images	8
Heaps	14
Stacks	5
PEBs/TEBs	4

Table 3. Entropy provided by ASLR for different program structures

The effectiveness of ASLR is heavily contingent on randomizing the base addresses of all program structures in the process' memory space. A talented attacker has the ability to bypass the overall protection offered by ASLR if the target program includes a single module that is not ASLR-enabled. Software developers can enable ASLR for their executable images by passing the `/DYNAMICBASE` option to VS linker (which is on by default since VS2008) or they can set this flag in the PE header of their binaries manually. The Windows loader checks the `DYNAMIC_BASE` flag, which is found in `DllCharacteristics` field of the PE header, for applying ASLR or not. Lately, Microsoft added a new feature at the registry level, "Force ASLR", for forcing applications to relocate images that are not built with the `/DYNAMICBASE` linker flag.

G. SUMMARY

This chapter has discussed the most prominent exploit mitigations that are incorporated into Microsoft products such as software development suites and operating systems. The availability of the discussed exploit mitigations in the latest versions of these products can be found in Appendix A. Figure 14 shows a timeline of the introduction of these mitigation technologies, both in Microsoft and outside. However, while these protections make exploit development harder and more unreliable, they also present weaknesses that enable talented attackers to bypass them (Sotirov & Dowd, 2008). Nevertheless, this does not mean that these protections are rendered totally ineffective. In the never-ending arms race between malicious attackers and defenders, there are two key points to consider: First, these mitigations are complementary to each other, so they are more effective when used together. Second, if any of these protections

is to be used, then all modules of the program should be protected, without exception. When applied together, these two factors obstruct most bypass techniques and, more importantly, increase the time and effort required to create reliable exploits.

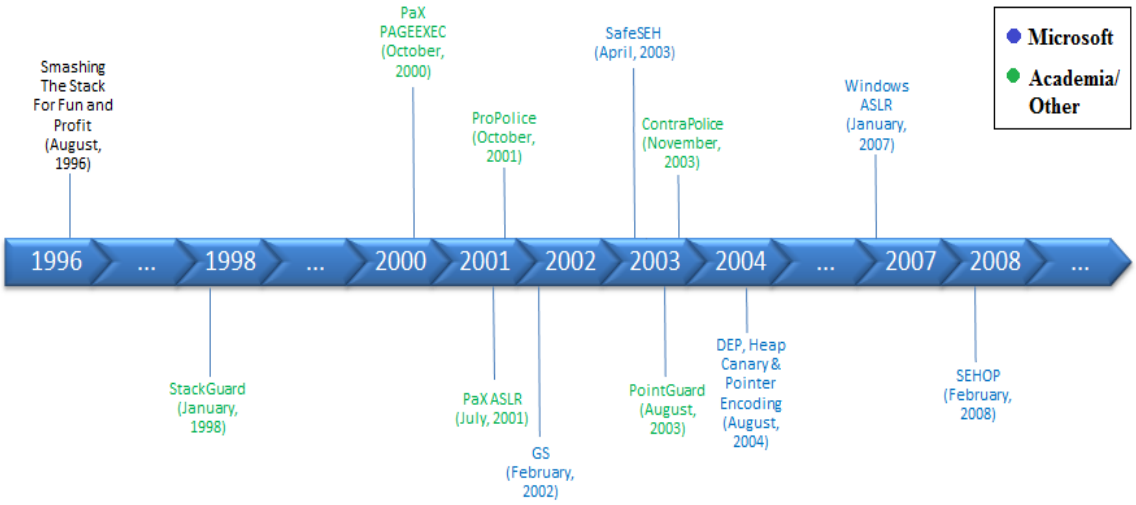

Figure 14. Timeline of exploit mitigations since Aleph One's paper

Since Gates' memo, Microsoft has covered a lot of ground in securing their products. Almost all applications that come with the latest consumer and server OSs, Windows 7 and Windows Server 2008, are protected with the mitigations discussed. However, the overall security of a system is also highly contingent on the security of third-party applications that are developed by myriad ISVs. The ISV adoption of mitigation technologies against memory corruption attacks represents an important part of the overall security posture of the system. In the next chapter, a wide range of third-party applications will be observed to depict and assess the implementation progress of these protections by ISVs.

IV. ANALYSIS OF SELECTED APPLICATIONS

In the previous chapter, various mitigation technologies offered by Microsoft were covered in detail. Microsoft ships all of its most recent OSs with software that has these mitigations enabled by default. However, it is clear that the overall security of a computer system not only depends on software that comes with the OS, but also software from other sources, such as ISVs. Therefore, adoption of these mitigations by ISVs plays a great role in making computer systems more secure against memory corruption attacks.

This chapter will present an analysis of forty-five applications that are widely used by end users all over the world, for implementation of these mitigations. First, the testing environment, selection criteria for applications, and scope will be covered. Then, methods used for validating the presence of these mitigations will be provided. Lastly, the results of the analysis will be presented.

A. ENVIRONMENT AND APPLICATION SELECTION

1. Selection of Operating System

According to Wikimedia Traffic Analysis Report (Zachte, 2012), which is based on monthly requests to Wikimedia pages,[12] as of April 2012, the Windows family of OSs constitute 74.67% of OSs that visited its pages. As seen in Figure 15, Windows 7 accounts for 38.73% of the worldwide page visits to Wikimedia websites, such as http://www.wikipedia.org. In the light of these statistics, it would be fair to say that Windows 7 is the most widely installed OS at the endpoints. Thus, Windows 7 SP1 was selected as the testing platform.

[12] Wikimedia operates online collaborative wiki projects including Wikipedia, Wiktionary, Wikiquote, etc. Wikipedia is among the top ten most-visited websites worldwide.

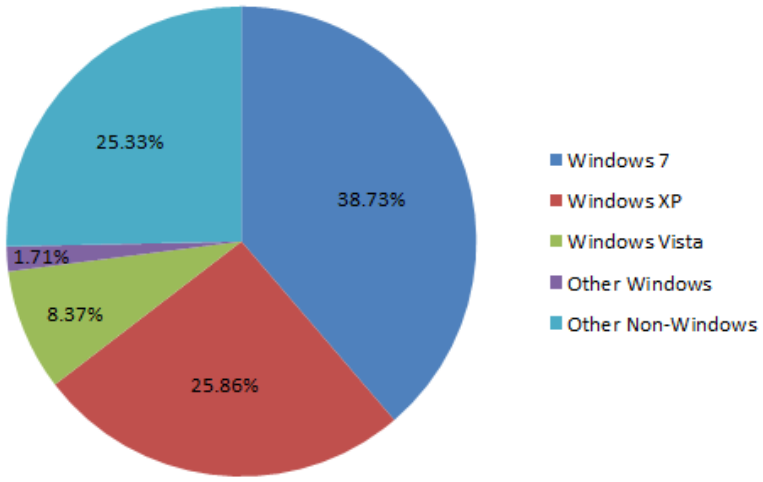

Figure 15. Market share of OSs based on Wikimedia statistics

2. 32-bit vs. 64-bit Architecture

Both for the OS and applications, 32-bit versions were selected over 64-bit versions. The underpinnings of this decision can be listed as:

- DEP is automatically turned on for 64-bit processes, so there is no need to check the availability of DEP in 64-bit applications.
- Termination of process on heap corruption is enabled by default on 64-bit processes, so `HeapSetInformation` API call is not needed.
- Exception handlers are no longer kept on the stack in 64-bit applications, so SafeSEH and SEHOP are not applicable for 64-bit processes.

For all these reasons and more (for example, ASLR is more effective in 64-bit architectures because of increased entropy), it would be reasonable to say that 64-bit applications running on 64-bit architectures are less vulnerable to successful exploitation than their 32-bit counterparts. More responsibility is left to software developers and ISVs when it comes to implementing these mitigations in 32-bit applications. That is why analysis was limited to 32-bit applications running on the 32-bit Windows 7 SP1.

3. Selection of Applications

For the selection of applications, nine distinct categories were determined, and for each category, five applications were selected for analysis. The categorization in general was based on well-known and widely used applications offered by Microsoft as part of their OSs, such as MS Office, Internet Explorer, Windows Media Player, etc. Applications were chosen among many alternatives based on popularity and wide user base. Another consideration during selection of the applications was the release date of these applications. Figure 16 shows the distribution of release dates of the selected applications.

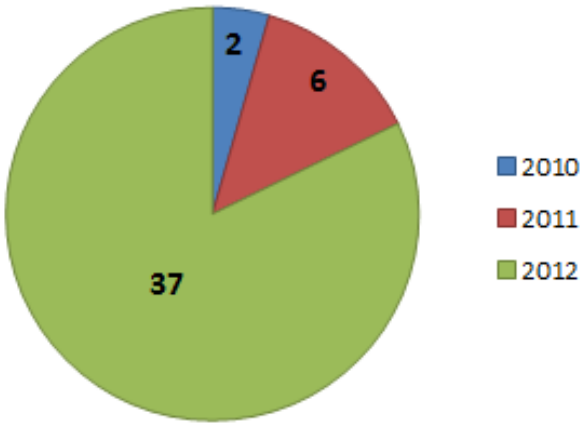

Figure 16. Distribution of the selected applications based on their release dates

Since the most recent mitigation introduced by Microsoft, SEHOP, dates back to 2008, ISVs had enough time (regarded as from two to four years), to implement these mitigations in their products. The categories and selected applications for each category are listed in Table 4. A more detailed list, which includes release dates and version info, is provided in Appendix B.

Another key point to note is that installers for some of the selected applications were found to contain extra software, such as browser toolbars. In this analysis, they were

considered part of their related applications, because most of the time users install these applications with default settings which confirm the installation of this extra software.

Category	**Applications**
Office Suites	Adobe Acrobat Reader, openOffice.org, Foxit Reader, AbiWord Word Processor, Nitro PDF Reader
Browsers	Firefox, Opera, Chrome, Safari, Maxthon
Media Players	Winamp, VLC Media Player, Quicktime Player, iTunes, Real Player
Instant Messaging Applications	mIRC, Yahoo Messenger, AIM, Nimbuzz, Trillian
Mail Applications	Thunderbird, MailCOPA, Zimbra Desktop, The Bat Home Edition, Spicebird
Antivirus Applications	Avast Antivirus, Norton Antivirus, McAfee Antivirus, AVG Antivirus, Kaspersky Antivirus
IE Plugins	Adobe Flash Player, Sun JRE, Google Toolbar, LastPass, Yahoo Toolbar
Download Managers	Download Accelerator Plus, Internet Download Manager, BitTorrent, iMesh, Flashget
Utilities	Winrar, Winzip, Nero, Babylon, Ccleaner

Table 4. Selected applications for analysis

B. METHODOLOGY

In order to test the selected applications for their implementation of the various mitigations described in Chapter III, three sources of information were checked. These are:

- The PE header of the executable (for SafeSEH, ASLR and DEP)
- The Windows Registry (for SEHOP)
- The binary contents of the executable file (GS, pointer encoding, and heap protection)

As touched upon a few times in the previous chapters, executable files in the Windows family of OSs have to be in a predefined format, i.e., portable executable (PE)

format. PE format is used to encapsulate information related to the executable file and is used by the Windows OS loader to manage these executable files. Files that adhere to PE format consist of headers and sections. One of these headers is called `IMAGE_OPTIONAL_HEADER` and has a 4-byte field named `DllCharacteristics`, which informs the loader about various settings for the executable file. The compatibility of each executable for ASLR and DEP is embedded in this four-byte value, so this field was checked in each executable file for verifying DEP and ASLR. `DllCharacteristics` field also encapsulates information that specifies whether an executable file includes SEH structures or not. Implementation of SafeSEH was checked using this data, as well as the presence of the safe exception handler table which is included in another part of PE header called the "Load Configuration Directory".

The Windows registry, where configuration settings and options for both OS and applications are stored, was checked for the implementation of SEHOP mitigation. SEHOP setting of an application (SEHOP setting is applied per application basis) was checked by verifying that "`Image File Execution Options`" registry key under "`HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNT\CurrentVersion\`" portion of the registry have the related application's "`DisableExceptionChainValidation`" value set to zero.

Pointer encoding mitigation which can be enabled by `EncodePointer` / `DecodePointer` API calls, heap corruption mitigation, which can be enabled by `HeapSetInformation` API call, and DEP, which can also be enabled at runtime by `SetDEPProcessPolicy` API call, were all checked by inspecting the binary contents of the executable file. This inspection included searching the names of these APIs in the strings dump of the executable file. Both import address table (IAT)[13] entries for these APIs, and calls made to these APIs by using `LoadLibrary` and `GetProcAddress` API pair, were captured using this inspection. Finally, implementation of stack canary mitigation (GS) was also checked by inspecting the binary contents of the executable file.

[13] A section in PE header that holds a lookup table for calling functions in different modules.

It was assumed that the file was protected by GS if any code pattern, like extra code added to a function's epilogue and prologue (shown in Figure 10), were found in the contents of the file. Also, any executable file protected by GS includes a code block for the initialization of the stack canary. This code block can easily be spotted in the contents of the binary file because it consists of five back-to-back API calls: `GetSystemTimeAsFileTime`, `GetCurrentProcessId`, `GetCurrentThreadId`, `GetTickCount`, and `QueryPerformanceCounter`. A random stack canary is generated by applying the XOR operation on the values returned from all these functions and used throughout the execution of the program.

For the inspection of all mentioned mitigations, three tools were used: PeStudio (Ochsenmeier, 2012), the strings utility from SysInternals Suite (Russinovich, 2012) and Recx SDL Binary Assurance for Windows (Whitehouse, 2012).

C. RESULTS

The analysis consisted of forty-five applications categorized into nine categories. A total of 8,012 files in PE format were inspected. Out of 8,012 files, 931 were .NET binaries (managed code), and thirty-two were 64-bit binaries. As the scope of this thesis does not include binaries that fall into these two categories (.NET and 64-bit), they were excluded from analysis. Among the remaining 7,049 files, 564 were directly executable files (executable files with an .exe extension). Finally, a total of 2,154 files were found to be in PE format, but did not include any code. These files are mostly used to store resources for programs, such as menus, pictures, and dialog templates. Table 5 summarizes these general descriptive statistics from the analysis.

Total Files Inspected	8012
Number of .NET binaries	931
Number of 64-bit binaries	32
Number of No-Code binaries	2154
Number .EXE binaries	564

Table 5. General statistics from the analysis

Before giving the adoption rates obtained in the analysis, it may be as well to point out an important finding. Three quarters of all of the analyzed applications (thirty-four out of forty-five) consisted of files from multiple vendors. This ratio is important because it shows that ISVs might not have full control over all of the files that are delivered as part of their products. Figure 17 provides more detailed insight about this observation.

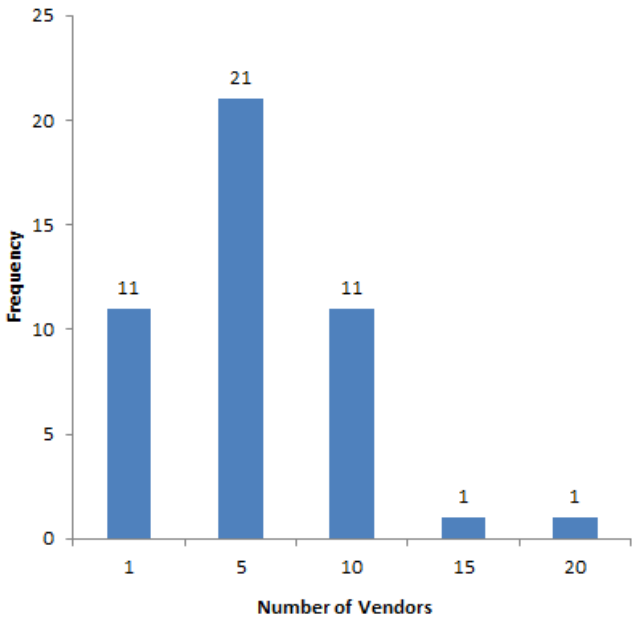

Figure 17. The distribution of vendor counts per analyzed application

The analysis showed that GS adoption is quite high among ISVs. Of the files checked, 93% are protected by GS, but of forty-five applications, only four have all of their executable files protected by GS. GS check was only applied to PE files that contain code and have a linker version associated with Microsoft Visual Studio products. Although the linker version does not guarantee that the file was compiled with the compiler that comes with that linker, it is mostly applied this way. Table 6 shows the distribution of linker versions and the number of GS protected files among inspected files.

Development Platform	Linker Version	Number of Files (contains code)	GS Protected Files
VS6	6.0	113	0
VS2002	7.0	9	1
VS2003	7.1	234	222
VS2005	8.0	1188	1085
VS2008	9.0	1641	1600
VS2010	10.0	1278	1244
Total		4463	4152

Table 6. Distribution of linker versions and GS adoption among inspected files

Variable reordering and parameter shadowing, which were discussed in the previous chapter, have been supported by the GS option starting with VS2005, so it is also important to note that GS protected files compiled with VS2002 and VS2003 are vulnerable to certain type of attacks that were mentioned in Chapter II.

SafeSEH adoption was also inspected for the same subset of files as in the GS option. The analysis showed that 91% of the files were protected by SafeSEH. Like GS mitigation, the number of applications that have all of their files protected by SafeSEH is only five out of forty-five. The situation is a little worse for SEHOP adoption: only two applications are protected. This should not be surprising, because SEHOP was introduced five years later than SafeSEH. Another reason for the low adoption of SEHOP is that it requires ISVs to enable it by modifying the registry, whereas SafeSEH has been enabled by default since VS2003.

Another mitigation technology that exhibited low adoption rate among ISVs was heap corruption mitigation. The `HeapSetInformation` API call was detected in only 126 files of the 4895 files that contained code. This corresponds to 2%. Furthermore, twenty-four applications out of forty-five did not have a single file that includes a call to

this API. This mitigation suffers from the similar problems as SEHOP. It was introduced in 2008, and requires ISVs to enable it by calling `HeapSetInformation` API programmatically.

Pointer encoding, which has been supported since 2004, was found to have a high adoption rate among ISVs. Roughly 77% of the files (3,809 files out of 4,895 files) contain references to `EncodePointer / DecodePointer` API pair. Of all the applications inspected, only one does not include any files that use this mitigation.

DEP was found to be enabled by only 65% of the directly executable files, which corresponds to 368 out of 564 files. DEP was inspected for directly executable files only, because it is a process-wide setting. Furthermore, nine out of forty-five applications have all of their files DEP enabled, while seven out of forty-five applications have none of their files DEP enabled.

Finally, the analysis showed that ASLR has relatively higher adoption rate than DEP. 76% of inspected executable files (5342 out of 7049 files) come with ASLR support. Application-wise, four out of forty-five applications come with all of their files ASLR enabled, and again four of them come with all of their files ASLR disabled.

In general, and not surprisingly, mitigations that are enabled by default, such as GS and SafeSEH, were found to have high adoption rates. DEP and ASLR, which are both prominent mitigations separately and further reduce the likelihood of a successful exploit when enabled together, were found to have room for improvement. Pointer encoding mitigation showed a high adoption rate, which is surprising as it requires extra API calls. SEHOP and heap corruption mitigation, both of which have been supported since 2008, showed that they need more time to become widely adopted. Figure 18 summarizes the overall situation by depicting all adoption rates from the analysis.

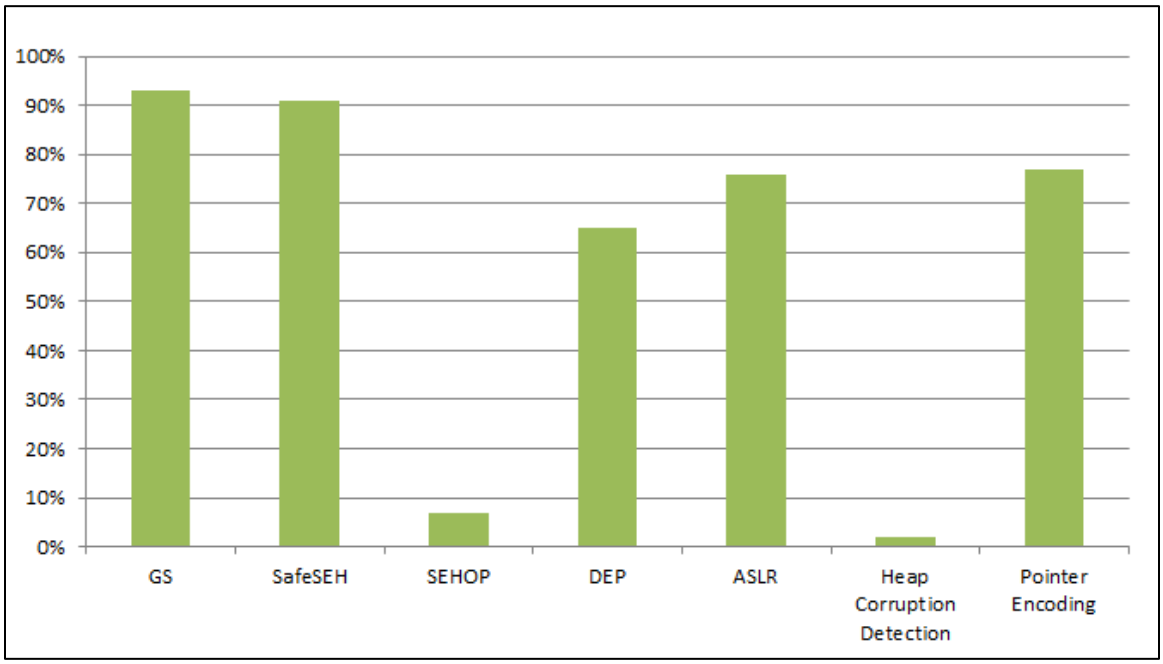

Figure 18. Adoption rates for all mitigations emerged from the analysis

V. CONCLUSION

In the last decade, global Internet usage has grown by 528%.[14] Increased use of mobile devices, such as cell phones and tablets, may boost this ratio in the upcoming years. In such a highly connected world, it is becoming more and more challenging to secure endpoints, and attackers enjoy an increasing number of potential victims. The biggest challenge in securing endpoints is the difficulty of controlling wide attack surfaces. Numerous third-party applications from many different ISVs with varying security postures broaden the attack surface for attackers.

The results of the analysis in Chapter IV showed that while most of the ISVs implement mitigations that are offered by Microsoft, it is hard to find any application that implements all these mitigations thoroughly. These mitigations are not silver-bullet solutions, especially when they are not implemented thoroughly and all together. There are various ways to bypass them singly, but when combined, it becomes harder and costlier for the attacker. Another interesting point observed in the analysis was that it takes time to adopt mitigations. It was seen that older mitigations have higher adoption ratios than newer ones. Also, as expected, making mitigations default results in higher adoption ratios.

This analysis shows that third-party applications coming from various ISVs pose greater risks to the security of endpoints than applications that come with operating systems. It would be wise for both simple home users and system administrators to keep track of the portfolio of third-party applications installed on their systems, assess these applications for their adoption of available mitigations, and make sure that they are updated periodically. Home users or system administrators should use tools such as

[14] Source: http://www.internetworldstats.com.

Microsoft's Enhanced Mitigation Experience Toolkit (EMET), which requires neither recompilation nor relink of applications, as a last resort to enforce some of the missing mitigations on critical applications.[15]

[15] Available at http://support.microsoft.com/kb/2458544.

APPENDIX

A. AVAILABILITY OF EXPLOIT MITIGATIONS

This appendix contains two tables: the first shows the availability of exploit mitigations in different versions of Windows family of OSs and the second one shows the availability of these mitigations in different versions of Visual Studio products.

		SafeSEH	SEHOP	Heap Protections	DEP	ASLR
Consumer Market	**XP RTM & SP1**	-	-	-	-	-
	XP SP2	+	-	Partial	Opt In	Partial
	XP SP3	+	-	Partial	Opt In	Partial
	Vista RTM	+	-	+	Opt In	Opt In
	Vista SP1 & SP2	+	Opt In	+	Opt In	Opt In
	Windows 7 RTM & SP1	+	Opt In	+	Opt In	Opt In
Server Market	**Server 2003 RTM**	-	-	-	-	-
	Server 2003 SP1 & SP2	+	-	Partial	Opt Out	Partial
	Server 2008 RTM	+	Opt Out	+	Opt Out	Opt In
	Server 2008 SP1	+	Opt Out	+	Opt Out	Opt In

	/GS	/SAFESEH	/NXCOMPAT	/DYNAMICBASE
VS6	-	-	-	-
VS2002	Opt Out	-	Opt In	-
VS2003	Opt Out	Opt Out	Opt In	Opt In
VS2005	Opt Out	Opt Out	Opt In	Opt In
VS2008	Opt Out	Opt Out	Opt In	Opt In
VS2010	Opt Out	Opt Out	Opt Out	Opt Out

B. THE LIST OF ANALYZED APPLICATIONS

This appendix contains the list of analyzed applications including each application's version number and release date.

No.	Application Name	Version	Release Date
1	Adobe Reader X	10.1.3	10.04.2012
2	OpenOffice	3.3.0	18.01.2011
3	Foxit Reader	5.1.4	06.01.2012
4	AbiWord	2.8.6	13.06.2010
5	Nitro PDF Reader	2.3.1.7	23.04.2012
6	Firefox	12.0	24.04.2012
7	Opera	11.62	27.03.2012
8	Chrome	19.0.1084.46	15.05.2012
9	Safari	5.1.5	26.03.2012
10	Maxthon	3.3.7.2000	19.04.2012
11	Winamp	5.6.2.3199	09.12.2011
12	VLC Media Player	2.0.1	19.03.2012
13	Quicktime Player	7.71.80.42	26.10.2011
14	iTunes	10.6.1.7	28.03.2012
15	Real Player	15.0.4.53	15.11.2011
16	Winrar	4.11	20.02.2012
17	Winzip	16.5	17.04.2012

18	Nero	11.2	29.02.2012
19	Babylon	9	26.03.2012
20	CCleaner	3.18.1707	25.04.2012
21	mIRC	7.22	13.10.2011
22	Yahoo Messenger	11.5	13.03.2012
23	AIM	7.5.12.6	05.03.2012
24	Nimbuzz	2.2.1	01.05.2012
25	Trillian	5.1.0.19	26.04.2012
26	Avast Antivirus	7.0.1426	07.03.2012
27	Norton Antivirus	19.1.0.28	02.05.2012
28	McAfee Antivirus	11.0.669.0	25.04.2012
29	AVG Antivirus	2012.0.2176	16.05.2012
30	Kaspersky Antivirus	12.0.0.374	04.05.2012
31	Adobe Flash Player	11.2.202.235	09.04.2012
32	Sun JRE	7.4	12.04.2012
33	Google Toolbar	7.3.2710.138	30.04.2012
34	LastPass	1.90.0	27.01.2012
35	Yahoo Toolbar	8.4.3.34	12.02.2012
36	Download Accelerator Plus	9.7	12.12.2011
37	Internet Download Manager	6.11	02.05.2012
38	BitTorrent	7.6.1	25.04.2012
39	iMesh	11.124124	29.04.2012
40	Flashget	3.7	16.03.2012
41	MailCOPA	12.01	21.05.2012
42	Thunderbird	12.0.1	30.04.2012
43	Zimbra Desktop	7.1.4	01.04.2012
44	The Bat Home Edition	5.1.2	23.04.2012
45	Spicebird	0.8	23.06.2010

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Abadi, M., Budiu, M., Erlingsson, U., & Ligatti, J. (2005). Control-Flow integrity: principles, implementations, and applications. *12th ACM Conference on Computer and Communications Security.* Alexandria: ACM.

Aleph One. (1996). Smashing the stack for fun and profit. *Phrack Magazine, 7*(49).

Alexander, S. (2005). Defeating compiler-level buffer overflow protection. *LOGIN, 30*(3), pp. 59–71.

Anderson, J. P. (1972). *Computer security technology planning study.* Air Force Systems Command. Retrieved February 9, 2012, from http://seclab.cs.ucdavis.edu/projects/history/papers/ande72a.pdf

Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., & Zovi, D. D. (2003). Randomized instruction set emulation to disrupt binary code injection attacks. *ACM Conference on Computer and Communications Security.* Washington D.C.: ACM Press.

Bhatkar, S., & Sekar, R. (2008). Data space randomization. *5th international Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Paris: Springer-Verlag.

Blexim. (2002). Basic integer overflows. *Phrack Magazine, 11*(60).

Bulba, & Kil3r. (2000). Bypassing stackguard and stackshield. *Phrack Magazine, 10*(56).

Chen, S., Xu, J., Sezer, E. C., Gauriar, P., Iyer, R. K., & Street, W. M. (2005). Non-control-data attacks are realistic threats. *Symposium A Quarterly Journal In Modern Foreign Literatures, 14*, 177–191.

Conover, M. (1999). *w00w00 on heap overflows.* Retrieved February 14, 2012, from http://www.cgsecurity.org/exploit/heaptut.txt

Cowan, C., Barringer, M., Beattie, S., & Kroah-Hartman, G. (2001). FormatGuard: automatic protection from printf format string vulnerabilities. *10th USENIX Security Symposium.* Washington D.C.: Usenix Association.

Cowan, C., Beattie, S., Johansen, J., & Wagle, P. (2003). PointGuard: protecting pointers from buffer overflow vulnerabilities. *Proceedings of the 12th USENIX Security Symposium* (pp. 91-104). Washington D.C.: USENIX Association.

Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., . . . Zhang, Q. (1998). StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. *Proceedings of the 7th USENIX Security Symposium.* San Antonio: USENIX Association.

Dobrovitski, I. (2003). Exploit for CVS double free() for Linux pserver. *Bugtraq Mailing List*.

Dowd, M., McDonald, J., & Schuh, J. (2006). *The art of software security assessment.*

Etoh, H., & Yoda, K. (2001). ProPolice: improved stack smashing attack detection. *IPSJ SIGNotes Computer Security, 14*.

Forrest, S., Somayaji, A., & Ackley, D. (1997). Building diverse computer systems. *Workshop on Hot Topics in Operating Systems* (pp. 62–72). Los Alamitos: IEEE Computer Society Press.

Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., & Wang, Y. (2002). Cyclone: a safe dialect of C. *USENIX Annual Technical Conference* (pp. 275-288). Monterey: USENIX Association.

Kiriansky, V., Bruening, D., & Amarasinghe, S. P. (2002). Secure execution via program shepherding. *11th USENIX Security Symposium.* San Francisco: USENIX Association.

Krennmair, A. (2003). *ContraPolice: a libc extension for protecting applications from heap-smashing attacks.* Retrieved from synflood.at/papers/cp.pdf

Lipner, S., Ladd, D., Simorjay, F., Pulikkathara, G., Jones, J., Miller, M., & Rains, T. (2011). *The SDL progress report.* Redmond: Microsoft.

Litchfield, D. (2003). *Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server*. Retrieved February 14, 2012, from http://dl.packetstormsecurity.net/papers/bypass/defeating-w2k3-stack-protection.pdf

Mitre. (2012). *Common vulnerabilities and exposures*. Retrieved from http://cve.mitre.org

Newsome, J., & Song, D. X. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *Network and Distributed System Security Symposium.* San Diego: The Internet Society.

Ochsenmeier, M. (2012). PeStudio (Version 3.69). Retrieved May 15, 2012, from http://www.winitor.com/

PaX. (2001). Retrieved from http://pax.grsecurity.net/

Pincus, J., & Baker, B. (2004). Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security and Privacy Magazine, 2*(4), pp. 20–27.

Rix. (2000). Smashing C++ VPTRs. *Phrack Magazine, 10*(56).

Russinovich, M. (2012). Sysinternals suite. Retrieved May 15, 2012, from http://technet.microsoft.com/en-us/sysinternals/bb842062

Scut. (2001). Exploiting format string vulnerabilities. Team Teso. Retrieved February 9, 2012, from http://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Secunia. (2012). *Secunia yearly report 2011*. Copenhagen: Secunia.

Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). *Proceedings of the 14th ACM conference on Computer and communications security* (pp. 552–561). New York: ACM.

Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., & Boneh, D. (2004). On the effectiveness of address-space randomization. *ACM conference on Computer and Communications Security* (pp. 298–307). Washington D.C.: ACM Press.

Solar Designer. (1997). "return-to-libc" attack. *Bugtraq Mailing List*.

Solar Designer. (2000). JPEG COM marker processing vulnerability. Retrieved from http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability

Sotirov, A., & Dowd, M. (2008). Bypassing browser memory protections in Windows Vista. *Proceedings of BlackHat.*

Spafford, E. H. (1989). Crisis and aftermath. *Communications of the ACM, 32*(6), pp. 678–687.

Suh, E., Lee, J., Zhang, D., & Devadas, a. S. (2004). Secure program execution via dynamic information flow tracking. *11th international conference on architectural support for programming languages and operating systems*. New York: ACM.

Tiobe. (2012). *Tiobe programming community index*. Retrieved from Tiobe software: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

van de Ven, A. (2004). *New security enhancements in Red Hat Enterprise Linux v.3, update 3.* Retrieved from http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf

Vendicator. (2000). *Stack Shield: a "stack smashing" technique protection tool for Linux.* Retrieved from http://www.angelfire.com/sk/stackshield/index.html

Wahbe, R., Lucco, S., Anderson, T. E., & Graham, S. L. (1993). Efficient software-based fault isolation. *14th ACM symposium on Operating systems principles.* New York: ACM.

Whitehouse, O. (2007). *An analysis of address space layout randomization on Windows Vista.* Cupertino: Symantec Corporation.

Whitehouse, O. (2012). Recx SDL binary assurance for Windows. Retrieved May 03, 2012, from http://www.recx.co.uk/exeaudit/index.php

Xu, W., Bhatkar, S., & Sekar, R. (2006). Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. *15th conference on USENIX Security Symposium.* Berkeley: USENIX Association.

Zachte, E. (2012). *Wikimedia traffic analysis report - operating systems*. Retrieved from Wikimedia Statistics: http://stats.wikimedia.org/wikimedia/squids/SquidReportOperatingSystems.htm

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Savunma Bilimleri Enstitusu
   Kara Harp Okulu, Bakanliklar, 06100
   Ankara, Turkey

4. Dan Boger
   Naval Postgraduate School
   Monterey. California

5. Chris Eagle
   Naval Postgraduate School
   Monterey. California