

A Comparison of Buffer Overflow Prevention Implementations and Weaknesses

Written by: Peter Silberman and Richard Johnson

Abstract

In the world of information security, buffer overflows remain the leading cause of software vulnerabilities. In recent years, the industry has seen an elevated rate of exploitation of these vulnerabilities due to readily available worm generation software and mass exploitation toolkits. This increasing exposure to buffer overflow attacks requires a technological solution that applies a protective layer against automated exploitation attempts.

This paper will examine two approaches to applying a generic protection against buffer overflow attacks and critique the effectiveness of available buffer overflow protection mechanisms on the Linux and Microsoft Windows platforms. An analysis of each technology will explain the methods by which a protection mechanism has been implemented and the technology's effectiveness in defending against automated attacks as well as targeted attacks, which specifically try to circumvent that specific protection method. Finally, a matrix will be presented which will define each technology's ability to protect against multiple classes of buffer overflow attacks including format strings, stack overflows and heap overflows.

1 Introduction

Software vulnerabilities which result in a stack based buffer overflow are not as common today as they once were. Unfortunately, it only takes a single known vulnerability in a commonly used piece of software or operating system to leave an entire infrastructure exposed. Since the release of papers detailing exploitation methods like Aleph1's "Smashing The Stack For Fun and Profit,"¹ Mudge's "How To Write Buffer Overflows"² and w00w00 "On Heap Overflows,"³ buffer overflows have been a prevalent problem in the information security field. The past few years has seen volumes of information published on techniques used to exploit software vulnerabilities. This research has become readily available at local bookstores, shortening the learning curve for an attacker even further. The availability of this information has led to the development of automated worms which can reduce the required attack window down to a number of hours before tens of thousands of computers are infected. In this sense, worm technology acts as a catalyst, which requires an equally effective defense against the attacks that a computer connected to the internet is subjected to on a daily basis.

1.1 Scope

This paper aims to explain the concepts behind buffer overflow protection software, implementation details of some of the more popular software in use, and provide an objective test platform which determines the effectiveness of each piece of software. The software covered by this paper includes PaX, StackGuard, StackShield, ProPolice SSP, Microsoft Visual Studio .NET, OverflowGuard, and StackDefender. The authors chose to omit Exec Shield⁴, kNoX⁵, RSX⁶ and OpenWall Project⁷ because the project ideas contributed to the formation of PaX. As a result their best features are covered in PaX and do not need to be explained twice.

¹ Smashing the stack for fun and profit (<http://secinf.net/uplarticle/1/p49-14.txt>), 2/19/04

² How To write buffer overflows (http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html), 2/19/04

³ w00w00 on Heap Overflow (<http://www.w00w00.org/files/articles/heaptut.txt>), 4/18/2021

⁴ Exec Shield (<http://people.redhat.com/mingo/exec-shield/>), 3/29/2004

⁵ kNoX (<http://isec.pl/projects/knox/knox.html>), 3/29/2004

⁶ RSX (<http://www.starzetz.com/software/rsx/>), 3/29/2004

⁷ OpenWall Project (<http://www.openwall.com/linux/>), 3/29/2004

2 Buffer Overflow Protection Technology

Buffer overflows can be addressed in a multitude of ways to protect against unwarranted code execution. The common implementations of these protection schemes have been separated into two categories: kernel enforced and compiler enforced protection.

2.1 Kernel Enforced Protection

Since the kernel is unaware of the internal functionality of the executable, its influence is restricted to modifications to the environment in which the program executes. The kernel is able to do this by modifying the layout of a process's virtual memory address space, as well as by applying access controls to pages of memory which prevent the execution of injected code. According to PaX, the goal of kernel enforced buffer overflow protection is to prevent and contain the following exploit objectives:

- Introduce/execute arbitrary code
- Execute existing code out of original program order
- Execute existing code in the original program order with arbitrary data

The two methods described below are combined to provide sufficient protection against most remote exploit attacks.

Memory Management Unit Access Control Lists (MMU ACLs)

Non-executable (NOEXEC) protection is the most commonly used access control for memory. A non-executable stack resides on a system where the kernel is enforcing proper "memory semantics." Proper memory semantics are comprised of three components outlined in the PaX Documentation⁸. One component is the separation of readable and writable pages, as well as only allowing programs that generate code at startup to have executable memory pages. The second component is to make all available executable memory including the stack, heap and all anonymous mappings non-executable. The third component consists of enforcing ACL, which involves denying the conversion of executable memory to non-executable memory and vice versa.

Address Space Layout Randomization (ASLR)

ASLR is based on the theory that exploits commonly rely on static values, such as addresses, which are known to contain operands such as 'call [register]' or pointers to the known location of a buffer on the stack. ASLR defeats these rudimentary exploit techniques by introducing randomness into the virtual memory layout for a particular process. ASLR can introduce varying levels of randomness during the process of loading a binary so that the binary mapping, dynamic library linking and stack memory regions are all randomized before the process begins executing. Randomizing the locations of the binary image, library locations, heap, and stack causes generic exploits to fail, requiring the exploit to brute force one or more address values and increasing the chance that an attack will be unsuccessful.

2.2 Compiler Enforced Protection

⁸ PaX Documentation (<http://pax.grsecurity.net/docs/pax.txt>), 4/18/2021

Compiler enforced protection mechanisms take a completely different approach to preventing the execution of arbitrary code within a protected process. Since the compiler has intimate knowledge of structure of the binary, modifications to the stack layout may be made. Introduction of special values called 'canaries' may be inserted into arbitrary points in memory to detect the corruption of saved control structures. The basic concept of overflowing a buffer to modify a return address or function pointer on the stack may be addressed by placing canary values in a location that would cause them to be overflowed before the return address may be reached. These canary values can be checked during the epilogue of a function, before a return to the saved pointer is made to ensure the integrity of the process control structures. In addition, modifications to the stack layout can ensure that a buffer overflow is unable to overwrite saved pointers by rearranging the order in which the variables are stored on the stack. More detail on stack layout modifications can be found in Section 3.2. Next we will take a closer look at stack canary values.

Stack Canaries

Stack canaries were first implemented by Immunix, Inc. (formerly known as WireX) in the StackGuard GCC patches. Preserving return addresses stored on the stack is the primary goal to prevent the redirection of code execution to an attacker controlled address space. The addition of a special canary value before the saved return address on the stack combined with a modification to the epilogue of a function, which checks the canary value, is an effective deterrent against arbitrary code execution.

There are four types of canaries that have been used to date:

Random Canary - The original concept for canary values took a 32-bit pseudorandom value generated by the /dev/random or /dev/urandom devices on a Linux operating system.

Random XOR Canary - The random canary concept was extended in StackGuard version 2 to provide slightly more protection by performing an XOR operation on the random canary value with the stored control data.

Null Canary - Originally introduced by der Mouse on the BUGTRAQ security mailing list, the canary value is set to 0x00000000 which is chosen based upon the fact that most string functions terminate on a null value and should not be able to overwrite the return address if the buffer must contain nulls before it can reach the saved address.

Terminator Canary - The canary value is set to a combination of Null, CR, LF, and 0xFF. These values act as string terminators in most string functions, and accounts for functions which do not simply terminate on nulls such as gets().

The use of canaries has been observed in three Linux compiler based protections as well as the Microsoft Visual C++ .NET compiler protections. The details of the Linux implementations may be found in Section 5.2 and an explanation of the .NET technology can be found in Section 6.1.

3 Attack Vector Test Platform

An attack vector test platform has been used in this paper to provide objective empirical data on the effectiveness of each protection mechanism. The test platform is based on work done by John Wilander for his paper titled "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention"⁹ and has been modified to compile on both Windows and Linux platforms. The attack vectors are defined by a combination of exploitation technique, location where the overflow occurs, and target value to overwrite. The techniques involved require the attack to overflow all the way to the target or overflow a pointer which redirects to the target. The locations are defined as the stack or heap/bss data segment. The attack targets include return address, saved base pointer, function pointer, and longjmp buffers. A complete listing of the test cases follows and will be referred to later in the paper.

1. Buffer overflow on the stack all the way to the target:
 - a. Return address
 - b. Old base pointer
 - c. Function pointer as local variable
 - d. Function pointer as parameter
 - e. Longjmp buffer as local variable
 - f. Longjmp buffer as function parameter
2. Buffer overflow on the heap/BSS/data all the way to the target:
 - a. Function pointer
 - b. Longjmp buffer
3. Buffer overflow of a pointer on the stack then pointing at target:
 - a. Return address
 - b. Base pointer
 - c. Function pointer as variable
 - d. Function pointer as function parameter
 - e. Longjmp buffer as variable
 - f. Longjmp buffer as function parameter
4. Buffer overflow of a pointer on the heap/BSS/data and then pointing at target:
 - a. Return address
 - b. Base pointer
 - c. Function pointer as variable
 - d. Function pointer as function parameter
 - e. Longjmp buffer as variable
 - f. Longjmp buffer as function parameter

⁹ A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention (http://www.ida.liu.se/~johwi/research_publications/paper_ndss2003_john_wilander.pdf), 2/7/03

4 Linux Protection Suites

There has been much work done on the Linux kernel and the GCC compiler to implement methods that prevent the exploitation of software vulnerabilities. This prior research has been instrumental in the design of new protection schemes for the Microsoft Windows operating system and various other hardened Linux projects. This section will explain the concepts of the most robust solutions currently available for Linux to provide context. In Section 5 an analysis of the Windows protections will be given.

4.1 Kernel Enforced Protection

We will use the PaX Project's kernel patches as an example of the most robust kernel-based protection software currently available. PaX offers prevention against unwarranted code execution via memory management access controls and address space randomization, referred to henceforth as NOEXEC and ASLR respectively. Section 4.1.1 outlines the components of NOEXEC and Section 4.1.2 will explain the design methods behind ASLR.

4.1.1 NOEXEC

The NOEXEC component of PaX aims to prevent the injection and execution of arbitrary code in an existing process's memory space. The NOEXEC implementation consists of three features which ultimately apply access controls on mapped pages of memory.

The first feature of NOEXEC applies executable semantics to memory pages. Executable semantics can be thought of as applying least privilege concepts to the MMU. The application of these semantics to create non-executable pages on the IA-32 architecture can take two forms, based on the paging (PAGEEXEC) and segmentation logic (SEGMEEXEC) of IA-32, and have tradeoffs between performance and usability. Once the logic required to create non-executable pages has been merged into the kernel, the next step is to apply the new features. This can be done by making the memory which holds the stack, heap, anonymous memory mappings and any section not specifically marked as executable in an ELF file, non-executable by default. Finally, the functionality of `mmap()` and `mprotect()` are modified to prevent the conversion of the default memory states to an insecure state during execution (MPROTECT). Each of these concepts is covered in more detail below.

PAGEEXEC

PAGEEXEC is an implementation of non-executable pages, which is derived from the paging logic of IA-32 processors. The IA-32 family of processors lack native hardware support for marking pages of memory non-executable. However, the implementation of a split Translation Lookaside Buffer (TLB) in Pentium and AMD K7+ CPUs can be leveraged to emulate non-executable page support. The purpose of the TLB is to provide a cache for virtual to physical address translation, which speeds up instruction or data fetching within the CPU. A split TLB actually has two separate translation buffers, one for instruction fetches (ITLB) and one for data fetches (DTLB). The ITLB/DTLB loading is the key feature to getting non-executable pages, as protected pages can be marked as either "non present" or "requiring supervisor level access." In both cases access to the pages will generate a page fault. The page fault handler can then decide if it was an instruction fetch or data access. If it

is an instruction fetch, it means that there was an execution attempt in a non-executable page, and the process can then be terminated accordingly. However, if the fault is triggered during data access, the pages can be changed temporarily to provide user level access and then restored to enable the fault handler for future accesses.

SEGMEXEC

SEGMEXEC is an alternate implementation of non-executable pages which is derived from the segmentation logic of IA-32 processors. Linux runs in protected mode with paging enabled on IA-32 processors, which means that each address translation requires a two step process. The logical address must first be converted to a linear address from which the correct physical address may be determined. This is usually transparent to users of Linux, primarily because it creates identical segments for both code and data access which cover the range of 0x00000000 - 0xffffffff and does not require translation between logical and virtual memory addresses because they share the same value. PaX leverages the segmentation logic to create separate address ranges for the data (non-executable) and code segments. The 3gb of userland memory space is divided in half and each segment is assigned one of the halves. The data segment lies in the 0x00000000 - 0x5fffffff range and the code segment lies in the 0x60000000 - 0xbfffffff range. Since the code and data segments are separated, accesses to the memory ranges can be monitored by the kernel and a page fault generated if instruction fetches are initiated in the non-executable pages.¹⁰

MPROTECT

MPROTECT is a feature of PaX which aims to prevent the introduction of new executable code to a given task's address space by applying access controls to the functionality of `mmap()` and `mprotect()`. The goal of the access controls is to prevent the following:

- Creation of executable anonymous mappings
- Creation of executable/writable file mappings
- Making executable/read-only file mapping writable except for performing relocations on an ET_DYN ELF file (non-PIC shared library)
- Conversion of non-executable mapping to executable

Every memory mapping has permission attributes which are stored in the `vm_flags` field of the `vma` structure within the Linux kernel. The four attributes which define the permissions of a particular area of mapped memory are `VM_WRITE`, `VM_EXEC`, `VM_MAYWRITE` and `VM_MAYEXEC`. The Linux kernel requires that `VM_MAYWRITE` is enabled if the `VM_MAYWRITE` attribute is true, and the same also applies to the `VM_EXEC` and `VM_MAYEXEC` attributes. Under normal operation, the Linux kernel can have a mapped area of memory with both write and exec permissions enabled, but the PaX Project must deny this combination to prevent the introduction of new code into executable pages. This reduces the number of possible states for memory permissions to be one of the following¹¹:

- `VM_MAYWRITE`
- `VM_MAYEXEC`
- `VM_WRITE | VM_MAYWRITE`
- `VM_EXEC | VM_MAYEXEC`

This essentially limits mapped memory to be either executable or writable, and ensures that both are never assigned at the same time. While these limits may break poorly designed

¹⁰ SEGMEXEC Documentation (<http://pax.grsecurity.net/docs/segmexec.txt>), 2/19/04

¹¹ MPROTECT Documentation (<http://pax.grsecurity.net/docs/mprotect.txt>), 2/19/04

software, it is an appropriate control to prevent the introduction of new code into executable areas of memory.

4.1.2 ASLR

Address Space Layout Randomization (ASLR) is the concept that attempts to render exploits which depend on predetermined memory addresses useless by introducing a certain amount of randomness to the layout of the virtual memory space. By randomizing the locations of the stack, heap, loaded libraries, and executable binaries, ASLR effectively reduces the probability that an exploit which relies on hardcoded addresses within those segments will successfully redirect code execution to the supplied buffer. Again, we will use PaX, which is comprised of four main components: RANDUSTACK, RANDKSTACK, RANDMMAP and RANDEXEC as our example implementation of ASLR.

RANDUSTACK

The RANDUSTACK component of PaX is responsible for randomizing userland stack addresses. The kernel is responsible for creating a program stack upon each `execve()` system call. This is done in a two step process which involves the kernel allocating the appropriate number of pages and populating them if necessary, and then mapping the allocated memory pages to the process's virtual address space. Typically on x86 architectures, the Linux kernel maps the stack at the end of the userland address space and grows downward from virtual memory address `0xbfffffff`. RANDUSTACK modifies addresses in both stages of the creation of the userland stack so that the kernel memory allocated and the virtual address mapping within the task are modified by a random value. It's noteworthy that the kernel addresses may shift by up to 4 kb while the userland stack may shift as much as 256 mb. It is also important to note that while forked processes will be handled by RANDUSTACK, threads within a process are randomized by the RANDMMAP component of PaX ASLR.

RANDKSTACK

The RANDKSTACK component of PaX is responsible for introducing randomness into a task's kernel stack. Each task is assigned two pages of kernel memory which is used to handle kernel mode operations during the lifetime of the task such as system calls, hardware interrupts and CPU exceptions. Normally, when the Linux kernel returns to userspace after a context switch to kernel mode during the execution of a system call or other operation, the kernel stack pointer will be at the point of initial entry to the kernel. This offers the advantage that the kernel stack pointer for a task may be randomized on each context switch rather than on each execution as is the case with userspace stack randomization. RANDKSTACK leverages this ability to randomize every system call; reasoning that every system call is a potential attack. While the amount of randomization that PaX adds to the kernel stack is limited to about a 128 byte shift. This should be enough to prevent the execution remote kernel exploits while keeping the assigned address sane.

RANDMMAP

RANDMMAP is the component which handles the randomization of all file and anonymous memory mappings. This is done in PaX by hooking the `do_mmap()` interface which is responsible for mapping the memory required for assigning `brk()` and `mmap()` managed heap space as well as executables and libraries. Note that only `ET_DYN` ELF executables are handled by RANDMMAP; `ET_EXEC` ELF executables are handled specifically by the

RANDEXEC component of PaX. RANDMMAP randomizes the specified memory mappings in two ways. The Linux kernel usually allocates heap space by beginning at the base of a task's unmapped memory and locating the nearest chunk of unallocated space which is large enough to supply the requested size. RANDMMAP modifies this functionality by adding a random `delta_mmap` value to bits 12-27 of the base address before searching for free memory. For executable mappings of ET_DYN binaries, the base load address is changed to that of a standard ELF_EXEC binary, which is located at 0x08048000 before the 16 bit `delta_mmap` value is added to introduce entropy.

RANDEXEC

The last major component of PaX is RANDEXEC. RANDEXEC is responsible for randomizing the location of ET_EXEC ELF binaries. The relocation of an executable that is not originally designed to be relocatable, raises some special concerns that are addressed by the RANDEXEC implementation. The first step is to load the executable at the standard address, but those pages must be marked non-executable. Next, a copy of the executable is created at a random location in memory using the same methods outlined in RANDMMAP. Again, the pages are mapped non-executable. However, the mirrored version may attempt to execute code and will be handled by the page fault handler. The page fault handler will perform some checks, including detecting if the page fault is due to an instruction fetch. It will then redirect the flow back into the randomized mapping by modifying the userland instruction pointer. These steps also account for hard coded addresses which are compiled into the binary since they will be handled by a page fault handler if the non-executable page is accessed instead of the randomly relocated image. PaX implements something called vma mirroring which handles the specifics of how addresses are translated when page fault handlers are in place. For brevity's sake we will not cover those details here. Further information can be found in the PaX documentation¹².

4.1.3 Defeating PaX

PaX offers considerable protection against buffer overflow attempts. Much research has been put into defeating PaX with little result. NOEXEC protections effectively prevent the execution of code on the stack, heap, and other data segments while the randomization of library addresses make return to libc exploitation much more difficult. However, research by Nergal¹³ in 2001 has shown that there are methods which may bypass the security of PaX. Since NOEXEC stack protection is difficult to circumvent, the attacker is forced to resolve randomized library addresses or use the PLT to resolve the function addresses for him. This may be done fairly simply locally, but remote exploitation may require an information leak vulnerability such as a format string bug to recover remote memory addresses. It may also be possible for an attacker to target a binary which is not compiled position independent. Such binaries may not be randomly `mmap()`'d which results in a standard return to libc exploitation scenario. The following results of the Attack Vector Test Platform show how well PaX protects a system against traditional attack vectors. It should be noted that this test was performed on a Linux 2.4 kernel with SEGMEXEC and all randomization functionality enabled.

PaX Attack Vector Test Platform Results

A plus symbol (+) indicates that the software successfully protected against the specified exploitation vector.

¹² VMA Mirroring Documentation (<http://pax.grsecurity.net/docs/vmmirror.txt>), 2/19/04

¹³ The advanced return-into-libc exploits: PaX case study (<http://www.phrack.org/phrack/58/p58-0x04>), 12/28/01

Buffer overflow on stack all the way to the target

- + Target: Parameter function pointer
- + Target: Parameter longjmp buffer
- + Target: Return address
- + Target: Old base pointer
- + Target: Function pointer
- + Target: Longjmp buffer

Buffer overflow on heap/BSS all the way to the target

- + Target: Function pointer
- + Target: Longjmp buffer

Buffer overflow of pointer on stack and then pointing to target

- + Target: Parameter function pointer
- + Target: Parameter longjmp buffer
- + Target: Return address
- + Target: Old base pointer
- + Target: Function pointer
- + Target: Longjmp buffer

Buffer overflow of pointer on heap/BSS and then pointing to target

- + Target: Return address
- + Target: Old base pointer
- + Target: Function pointer
- + Target: Longjmp buffer

4.2 Compiler Enforced Protection

The compiler based approach to preventing the exploitation of buffer overflow vulnerabilities is primarily based on ensuring the integrity of control data stored on the stack. There have been three major implementations of compiler based protections which are all based on modifications to the stack layout and/or the use of canaries. This section will take a deeper look at the specific differences in the implementations between the StackGuard, StackShield and Stack-Smashing Protector (previously known as ProPolice) buffer overflow protection mechanisms.

4.2.1 StackGuard

StackGuard is a GCC patch created by Immunix, Inc. which has provided the foundation for other compiler based protection technologies and pioneered the use of stack canaries as a method for preventing the overwriting of saved control values. The StackGuard patch adds code at the RTL level to the `function_prologue` and `function_epilogue` functions within GCC to provide the generation and validation of the stack canary. StackGuard originally modified the `function_prologue` to make GCC push a random canary directly before the return address. The most recent version has been modified to protect the saved registers and frame pointer in addition to the return address and implements terminator canary values. The placement has been modified from its original location right before the saved return address on the stack to be placed in a location which is harder to overwrite. The decision of where to place the canary is architecture specific. On x86 architectures, the saved frame pointer points to a location where alignment padding has been generated by GCC and provides a good address to store the canary. The stored canary will be checked by the

function_epilogue before a function may return. If the stored canary does not match the canary on the stack, StackGuard will exit the program and record an error in the system log.

Defeating StackGuard

While StackGuard may effectively stop standard stack overflows which overwrite a saved return address, there are other attack vectors which may easily bypass the canary check. Since the day after StackGuard was originally released, methods for bypassing StackGuard protected binaries have been publicly discussed.¹⁴ Tim Newsham explored the possibility of bypassing StackGuard by overwriting local variables which could then be used to compromise the protection. Additional research has shown that overwriting function pointers and frame pointers stored on the stack can also lead to compromise¹⁵. Protection against non-stack based attack vectors such as heap overflows is also beyond the scope of StackGuard. The reader may view the results of the Attack Vector Test Platform below to better understand the protection coverage provided by StackGuard.

StackGuard Attack Vector Test Platform Results

A plus symbol (+) indicates that the software successfully protected against the specified exploitation vector.

Buffer overflow on stack all the way to the target

- Target: Parameter function pointer
- Target: Parameter longjmp buffer
- + Target: Return address
- + Target: Old base pointer
- Target: Function pointer
- Target: Longjmp buffer

Buffer overflow on heap/BSS all the way to the target

- Target: Function pointer
- Target: Longjmp buffer

Buffer overflow of pointer on stack and then pointing to target

- Target: Parameter function pointer
- Target: Parameter longjmp buffer
- Target: Return address
- + Target: Old base pointer
- Target: Function pointer
- Target: Longjmp buffer

Buffer overflow of pointer on heap/BSS and then pointing to target

- Target: Return address
- + Target: Old base pointer
- Target: Function pointer
- Target: Longjmp buffer

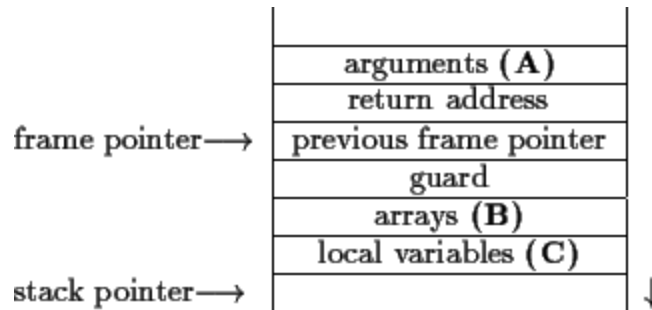
ProPolice Stack-Smashing Protection (SSP)

Unlike other compiler methods, which just place canaries in front or behind the return address, SSP proactively monitors stack changes. SSP re-arranges argument locations, return addresses, previous frame pointers and local variables. SSP has come up with the

¹⁴ Re: StackGuard (<http://online.securityfocus.com/archive/1/8260>), 12/19/97

¹⁵ Different tricks to bypass StackShield and StackGuard protection (<http://www2.corest.com/files/files/11/StackguardPaper.pdf>), 6/3/02

following “safe stack model” that helps decide where variables, arguments and the canaries should be placed on the stack:



As the reader can see from the figure above, the array and local variables are all below the return address. If an overflow were to occur in the array, nothing important would be overwritten and the overflow would be useless. This also helps with pointers. Take a look at a vulnerable code segment (provided by SSP documentation):

```
void bar( void (*func1)( ) )
{
    void (*func2)( );
    char buf[128];
    .....
    strcpy (buf, getenv ("HOME"));
    (*func1)( ); (*func2)( );
}
```

Without our model, an overflow in buf could overwrite the function pointers. However, SSP will change this code to:

```
void bar( void (*tmpfunc1)( ) )
{
    char buf[128];
    void (*func2)( );
    void (*func1)( ); func1 = tmpfunc1;
    .....
    strcpy (buf, getenv ("HOME"));
    (*func1)( ); (*func2)( );
}
```

Here the reader will see that by following the SSP safe stack diagram, the passed function pointer is put in a register if possible. If there is no register available, SSP puts the function pointer in a local variable, making it safe. The re-arrangement will, in the case of an overflow, overwrite nothing important.¹⁶

Defeating ProPolice SSP

¹⁶ Stack Protection Method (<http://www.tri.ibm.com/projects/security/ssp/node4.html>), 2/19/04

ProPolice has proved to offer much better protection against stack overflows than the other compiler patches, yet inherent design flaws still leave certain attack vectors exposed. ProPolice does not protect arrays with less than eight elements. An overflow of a small buffer will go unchallenged and may redirect the return address to shellcode stored elsewhere in memory. An additional design limitation may leave members of structures unprotected as well since the reordering of variables within the structure is not possible. Standard attacks which leverage a pointer overwrite to control arbitrary memory locations may be used if the pointer is contained within a structure. Again, the protecting the heap is not within the scope of the ProPolice software so it is not expected that those attacks will be defeated. An overview of the protection provided by ProPolice is presented below.

ProPolice SSP Attack Vector Test Platform Results

A plus symbol (+) indicates that the software successfully protected against the specified exploitation vector.

Buffer overflow on stack all the way to the target

- + Target: Parameter function pointer
- Target: Parameter longjmp buffer
- + Target: Return address
- + Target: Old base pointer
- + Target: Function pointer
- + Target: Longjmp buffer

Buffer overflow on heap/BSS all the way to the target

- Target: Function pointer
- Target: Longjmp buffer

Buffer overflow of pointer on stack and then pointing to target

- + Target: Parameter function pointer
- + Target: Parameter longjmp buffer
- + Target: Return address
- + Target: Old base pointer
- + Target: Function pointer
- + Target: Longjmp buffer

Buffer overflow of pointer on heap/BSS and then pointing to target

- Target: Return address
- + Target: Old base pointer
- Target: Function pointer
- Target: Longjmp buffer

4.2.2 StackShield

While development on StackShield has appeared to cease, we will cover the basic concepts here as an additional approach to compiler based protection. StackShield is similar to the other compiler-based protections, but also has some unique features. The first feature is the Global Return Stack, which acts as a specialized stack for return addresses. Every time a function is called, the return address is copied to the Global Ret Stack. When a function is ready to return, the return address is copied from the Global Ret Stack to the applications stack, overwriting any possible compromise. Since this method will not detect attacks, the Ret Range Check feature may be used instead, which copies the return address to an unwriteable area rather than pushing a canary on the stack during the function prologue. When function epilogue is reached, StackShield will check the stored return address. If an inconsistency is found, StackShield will exit the program, and allow for the detection and logging of overflow attempts. StackShield also offers protection of function pointers. This method is simple, but breaks programs that allocate memory dynamically. StackShield's protection of function pointers only allows function pointers to point to the .text section since any injected code would have to be in the .data section. This method completely denies attempts to run malicious code that is unable to overwrite the .text segment.

Defeating StackShield

Methods for defeating StackShield are similar to those used to bypass StackGuard protection. The use of the Global Return Stack does provide some additional protection against pointer overwrites that StackGuard does not offer. Pointer overflows which later

modify the return address directly on the stack will fail since the modified value is overwritten when the saved return is restored from the Global Return Stack. Other than this one exception, pointer overwrites may still be used to execute arbitrary code by controlling saved function pointers or overwriting GOT entries. The similarities in protection coverage to StackGuard can be seen in the results of the Attack Vector Test Platform.

StackShield Attack Vector Test Platform Results

A plus symbol (+) indicates that the software successfully protected against the specified exploitation vector.

Buffer overflow on stack all the way to the target

- Target: Parameter function pointer
- Target: Parameter longjmp buffer
- + Target: Return address
- + Target: Old base pointer
- Target: Function pointer
- Target: Longjmp buffer

Buffer overflow on heap/BSS all the way to the target

- Target: Function pointer
- Target: Longjmp buffer

Buffer overflow of pointer on stack and then pointing to target

- Target: Parameter function pointer
- Target: Parameter longjmp buffer
- + Target: Return address
- + Target: Old base pointer
- Target: Function pointer
- Target: Longjmp buffer

Buffer overflow of pointer on heap/BSS and then pointing to target

- + Target: Return address
- + Target: Old base pointer
- Target: Function pointer
- Target: Longjmp buffer

5 Windows 2003 Stack Protection

5.1 Windows 2003 Stack Protection

5.1.1 Compiler-based Protection

Microsoft began looking for ways to ensure that their products were secure out of the box. This decision was followed by the release of multiple vulnerabilities in the summer of 2003. Microsoft's solution to making their products secure out of the box, is very similar to Crispin Cowan's StackGuard covered earlier in this paper. In the new .NET compilers, Microsoft provides the /GS command line switch. When enabling this command line switch, a security cookie (canary), is placed in front of the return address and saved ebp. By default, Windows 2003 is compiled with it.

5.1.2 How The Protection Works

When a program that is compiled with the /GS switch, returns from a function. The canary authentication mechanism loads the canary that was on the stack into ecx and compares it to the original canary, that is stored in the .data section of the program. If these canaries match each other the program continues. However if these canaries do not match, the program checks to see if a security handler is specified in the .data section of the program. If a security handler is specified the program calls that security handler. However if no security handler is specified then the UnHandledExceptionFilter is set to 0x00000000 and called. The UnHandledExceptionFilter will load faultrep.dll and call an exported function named ReportFault.

5.1.3 Compromising The Protection

There are many ways outlined by researchers namely David Litchfield on how to bypass Microsoft's protection. This paper will outline briefly how it can be done.

If a local buffer is overflowed, the attacker may try to overwrite the EXCEPTION_REGISTRATION structure and within that overwrite the exception handler. The EXCEPTION_REGISTRATION structure looks like:

```
EXCEPTION_REGISTRATION
Pointer to next structure on the stack
Pointer to the exception handler
```

Microsoft recognized that the exception handler was being abused in attacks. Microsoft came up with an outline to prevent this from happening. All registered exception handlers are stored in the program's Load Config Directory. If the exception handler is not in the Load Config Directory array the exception handler is not called. The exception handler is also not called if the handler points to the stack. This is where Microsoft's defense stops against abusing the exception handler stops. Microsoft allows exception handlers that point outside the modules range to be called and Microsoft also allows exception handlers that point to the heap to be executed.

David Litchfield outlines two reliable avenues for attack which will be discussed. The first attack method is to point the exception handler to a registered exception handler and abuse the registered exception handler. The second method is to point the exception handler to a code block outside the address range that will when executed point back into the attacker's code.

After the attacker has compromised the exception handler they only need to cause an exception. This can be done by either causing a memory access violation by forcing the program to read or write to an address that does not exist. Or to write passed the end of the stack causing an exception.

There are more less reliable attack avenues discussed by David Litchfield. However this subject has been talked about before at conferences and in multiple books therefore this ends our discussion of Windows 2003 stack protection and to find more information we supply you with links.

- <http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf>
- Shellcoders Handbook, pg 161-167

5.1.4 Bypass Windows 2003 Stack Protection

Windows 2003 Stack Protection Attack Vector Test Platform Results

A plus symbol (+) indicates that the software successfully protected against the specified exploitation vector.

```
Buffer overflow on stack all the way to the target
+ Target: Parameter function pointer
+ Target: Return address
+ Target: Function pointer
```

```
Buffer overflow of pointer on stack and then pointing to target
+ Target: Parameter function pointer
+ Target: Return address
+ Target: Function pointer
```

5.2 NGSEC StackDefender 1.10

5.2.1 StackDefender Overview

StackDefender offers driver based stack protection. Previously in the paper this method was known as Kernel Enforced protection. StackDefender is able to offer protection against malicious code execution because StackDefender monitors specific API calls and checks these API calls to make sure they are not made from the stack. StackDefender installs a driver called StackDefender.sys that implements a hooking solution known as "NT System-Call Hooking"¹⁷ ¹⁸ (see brief Description of Kernel System-Call Hooking). StackDefender hooks both ZwCreateFile and ZwOpenFile by replacing the KeServiceDescriptorTable address that point them. By hooking these system functions, StackDefender can see all files opened and created. When either ZwCreateFile or ZwOpenFile are called, the driver looks to see if the file executed is one of the following: msvcrt.dll, ntdll.dll, or kernel32.dll. If it is one of these files, the driver replaces the last six characters with NG.fer, this will then load SD's

¹⁷ Mark Russinovich and Bryce Cogswell, "Windows NT System-Call Hooking", Dr. Dobb's Journal January 1997

¹⁸ Windows NT System Service Table Hooking (<http://www.wiretapped.net/~fyre/sst.html>), 5/11/04

own version of these DLL's. The DLL's that StackDefender loads have all been rebased, using Microsoft's ReBaseImage¹⁹ API.

5.2.2 Brief Description of Kernel System-Call Hooking

Before one can understand how Kernel System-Call Hooking works they must understand how a kernel system-call is made. The system call is made through the int 2e handler which is internally called KiSystemService. Before the int 2e instruction is executed, certain registers must be filled in. The eax register must contain the service id number, and edx must contain a frame pointer to the userland stack where the parameters are stored, so that the kernel can copy the parameters off the userland stack and onto the kernel stack. Once the int 2e instruction is executed, the processor switches to kernel mode and executes the int 2e handler. The int 2e handler indexes the ServiceTableBase member of the structure KeServiceDescriptorTable. The structure itself looks like:

```
typedef struct ServiceDescriptorTable {
    PVOID ServiceTableBase;
    PVOID ServiceCounterTable(0);
    unsigned int NumberOfServices;
    PVOID ParamTableBase;
}
```

Here is an example of a Kernel System-Call:

```
__asm
{
    mov eax, 0x64 //System Service ID (Function Number To Call)
    lea edx, [esp+0x04] //Pointer to parameters on stack
    int 2eh //switch to kernel mode and execute handler
}
```

This gets translated into the following, a pseudo code example will show how the kernel calls the system call:

```
call KeServiceDescriptorTable->ServiceTableBase[function_id]
```

The hook itself is implemented by a driver, simply overwriting the pointer at ServiceTableBase + function_id to point to the driver's version of that function. The driver can then call the original function when it is done.

5.2.3 Individual Analysis of files

5.2.4 StackDefender.sys

StackDefender.sys makes up half of the NGSEC StackDefender protection suite. StackDefender.sys is the component that performs the system-call hooking and file redirection.

StackDefender.sys has two phases. The first phase is to setup a system-call hooks on both ZwCreateFile and ZwOpenFile. These hooks force any file opened or created to go through StackDefender's driver.

¹⁹ MSDN ReBaseImage (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/rebaseimage.asp>), 5/11/2004

StackDefender's second phase occurs when files are opened/created. StackDefender looks at the file being created/opened if the file is either msvcrt.dll, ntdll.dll or kernel32.dll. StackDefender will overwrite the last 6 bytes of the filename with NG.fer this turns the file name into, msvcNG.fer, ntdNG.fer and kernelNG.fer. The *NG.fer files are put on the system during the installation process, so they do exist and are successfully loaded into the process space.

5.2.5 stackdefender_service.exe

stackdefender_service.exe, installs and creates the service on the system the user wants to protect. The service executable registers StackDefender.sys as the service's driver, so that the driver can modify the KeServiceDescriptorTable. Finally, the executable interacts with the Service Control Manager (SCM) to stop start and disable the service as the user wants.

5.2.6 Background NG.fer files

During the installation of StackDefender, StackDefender will place three NG.fer files on the user's system. Two of the three NG.fer files that are installed by StackDefender are replicas of the DLL's they replace. During the installation processes StackDefender will copy msvcrt.dll, and ntdll.dll into files called msvcNG.fer and ntdNG.fer. These two new files are then rebased (see section on What ReBasing Is). The rebase process tries to give the DLL's a random image base, using an algorithm designed to come up with random numbers.

5.2.7 What ReBasing Is

When a DLL is loaded into the memory of a process via LoadLibrary, or other similar API function, the system will load the DLL at the preferred address range, which is known as the image base specified in the PE Optional Header section. If the memory is taken up, and the DLL has .reloc information the DLL is reloaded into a different address space. However, if the DLL is a system DLL (e.g. ntdll.dll), these DLL's are not relocatable on the fly. Thus if the preferred load address is taken for a system DLL, the application will not be loaded.

NGSEC StackDefender, is able to load the DLL into a new memory region, by creating a copy of the DLL and rebasing the copy of that system DLL. NGSEC StackDefender uses an API known as ReBaseImage²⁰ documented in the MSDN library to rebase the three DLL's it replaces.

5.2.8 kernelNG.fer

The kernelNG.fer is different from the other two NG.fer files because it is not just a copy of the DLL from the users system that was rebased. StackDefender's first phase in modifying kernelNG.fer, is to modify the relocation section (.reloc). StackDefender changes .reloc section flags from 42000040 (Readable + Discardable + Initialized Data) to E2000060 (Executable + Writable + Readable). StackDefender then hooks functions that the NGSEC believes will be used in shellcode. NGSEC believes that by watching any functions that will

²⁰ ReBaseImage (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/rebaseimage.asp>), 5/12/04

be used by shellcode, they can prevent the success of shellcode. The following functions are hooked using a method called "Export Address Table Relocation":

- WinExec
- CreateProcessA
- CreateProcessW
- CreateThread
- CreateRemoteThread
- GetProcAddress
- LoadModule
- LoadLibraryExA
- LoadLibraryExW
- OpenFile
- CreateFileA
- CreateFileW
- _lopen
- _lcreat
- CopyFileA
- CopyFileW
- CopyFileExA
- CopyFileExW
- MoveFileA
- MoveFileExW
- MoveFileWithProgressA
- MoveFileWithProgressW
- DeleteFileA
- LockFile
- GetModuleHandleA
- VirtualProtect
- OpenProcess
- GetModuleHandleW

This method of hooking is very different from the system-call hooking method that StackDefender.sys implements. Instead this implementation modifies the Export Address Table (EAT) so that the hooked function's EAT entries point into the .reloc section where the new function's handling code is contained. The EAT is a structure which contains the entry points to API's that are contained in the DLL. The code put in the .reloc section is the same for each function. The code will load proxydll.dll and then call a function exported in proxydll.dll this exported function's name is StackDefender. Proxydll.dll will decide whether or not the application has been compromised. Read the next section about proxydll.dll to see how it is done.

5.2.9 proxydll.dll

Proxydll.dll is the other half of the overflow protection suite that NGSEC offers. Proxydll.dll plays a pivotal role in the detection of malicious code execution. KernelNG.fer hooks certain API functions that NGSEC believes will be used in the average shellcode (such as CreateProcessA or LoadLibrary). When a program calls one of these hooked functions, the function loads and calls proxydll.dll. Proxydll.dll exports one function which is called StackDefender. StackDefender takes four parameters. All four of these parameters are integers. The paper will refer to them as arg1, arg2, arg3, and arg4. These arguments are as follows, arg1 is [esp + 0x0C], arg2 address from where API is called, arg3 is single integer

and arg4 is the stack address of a given parameter that was fed into that specific API. The key parameters that the reader should focus on are arg1 and arg2. These two parameters are how StackDefender decides if the program has been compromised. What occurs after StackDefender has gotten these arguments is significant. StackDefender will call a new function that takes arg4 as a parameter. StackDefender will go to the address of arg4 (where the parameter was originally from), there StackDefender will search the stack for strings that commonly exist in shellcode string tables. The list of strings are as follows (quotes are put around so the reader can see the exact string format):

```

“cmd.exe”
“net “
“LoadLibrary”
“LoadModule”
“CreateProcess”
“WinExec”
“CreateThread”
“GetModuleHandle”
“OpenProcess”
“OpenThread”
“CreatePipe”
“bind”
“connect”
“wsock32”
“wsock32” (Unicode)
“cmd.exe” (Unicode)
“net “ (Unicode)

```

String tables are used in basic shellcode. The reason being that a lot of buffer overflows are the result of the misuse of string manipulation calls. If a string function such as strcpy or strcat receives a NULL (00), the string function will terminate the string at the NULL and might not reach the attackers shellcode. To prevent this from happening, DilDog came up with an ingenious idea. He decided to create something called a string table. The string table will store all the shellcode's strings. The string table will then be xor with a number to create a string table that has no NULLs (00). This method requires the attacker to add a little code to the front of their shellcode a couple commands that will loop through the xor'd string table and un-xor putting the strings on the stack so that the shellcode can use them. This process is discussed in detail by DilDog of Cult of the Dead Cow²¹. What StackDefender has done is essentially decided to search the stack for certain strings that are commonly in string tables. However in looking at the program it does not exit if a string is found on the stack. It continues to the next phase.

The next phase in the detection process of malicious code execution is checking where the functions were called from. NGSEC designed an algorithm that helps detect if a program was compromised. After the checking for the existence of the string table, StackDefender then checks where the API call originated from. StackDefender does this by calling VirtualQuery²² on arg1 and arg2. Arg1 is the stack address 0x0C from the esp, and arg2 is the address from where the API is called. If arg1's page allocation base is equal to arg2's page allocation base, then the function call is coming from the stack. The page allocation base can be determined by looking at the MEMORY_BASIC_INFORMATION²³ structure that is passed into

²¹ The Tao Of Windows Buffer Overflow (http://www.cultdeadcow.com/cDc_files/cDc-351/index.html), 5/14/04

²² VirtualQuery (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualquery.asp>), 5/14/04

²³ MEMORY_BASIC_INFORMATION (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/memory_basic_information_str.asp), 5/14/04

VirtualQuery. This tells the program that the calls are on the stack. Therefore StackDefender marks the program as exploited and exits it. The second check StackDefender performs before marking the program safe, is to check if the callers address space is writeable. If the API call is made from an executable, the executables image has been VirtualProtected²⁴ so that the write flags are off. StackDefender is able to check this by calling IsBadWritePtr²⁵. If IsBadWritePtr returns zero, it means that the calling address is writeable and StackDefender will mark it as an application that has been overflowed, and exit accordingly. If all these checks pass, StackDefender will let the application execute normally without much cost to system resources and speed.

5.2.10 Defeating StackDefender

While StackDefender in the standard Attack Vector Test Platform performed the best out of third party protection suites. It is an illusion to think StackDefender 1.10 is the best. The reason StackDefender 1.10 did so well in our tests is that the Attack Vector Test Platform uses standard shellcode that any attacker would use (the shellcode used is taken from Matt Millers paper on win32 shellcode²⁶). However if an attacker knew the system were protected with StackDefender 1.10 the attacker could simply write their own GetProcAddress and LoadLibrary functions that do not call the hooked API's. The attacker could then call ZwAllocateVirtualMemory, memcpy, ZwProtectVirtualMemory to allocate memory write shellcode to the memory and then protect the memory so that StackDefender does not know the memory was at one point writable.

However simplistic this protection scheme is, the protection scheme none the less proved to offer the best protection for third party applications. Below are the Attack Vector Test Platform results to help the reader better understand how well this application protects third party programs.

StackDefender 1.10 Attack Vector Test Platform Results

A plus symbol (+) indicates that the software successfully protected against the specified exploitation vector.

Buffer overflow on stack all the way to the target

```
+ Target: Parameter function pointer
+ Target: Return address
+ Target: Function pointer
```

Buffer overflow of pointer on stack and then pointing to target

```
+ Target: Parameter function pointer
+ Target: Return address
+ Target: Function pointer
```

5.3 OverflowGuard

²⁴ VirtualProtect (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualprotect.asp>), 5/14/04

²⁵ IsBadWritePtr (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/isbadwriteptr.asp>), 5/14/04

²⁶ Win32-Shellcode (<http://www.hick.org/code/skape/papers/win32-shellcode.pdf>), 7/6/04

5.3.1 OverflowGuard Overview

OverflowGuard is a protection suite that claims to offer “strong Unix style buffer overflow protection for windows.” OverflowGuard is a driver based memory protection suite. This protection method is known as Kernel Enforced. OverflowGuard is based on the PaX Project²⁷ document, and attempts to offer similar PaX like protection to windows users. OverflowGuard is able to offer this kind of protection by performing a number of kernel level operations before the user ever executes a program. OverflowGuard modifies the Interrupt Descriptor Table (see Interrupt Descriptor Table (IDT) and Other Terms for more information), to point to its own handlers for certain exceptions. OverflowGuard then causes pages faults by setting the bit 1 or 2 of the Page Table Entry to zero. This marks the page as read-only or supervisor, and any service writing/accessing the page will fault. These faults will then execute OverflowGuard’s handlers and allow OverflowGuard to examine the instruction and determine if the program has been compromised or not.

5.3.2 Interrupt Descriptor Table (IDT) and Other Terms

5.3.3 IDT Description

An IDT exists on most Pentium processors and on all x86 systems . This table is used by both the processor and operating system (OS). The processor/OS will look up the IDT table and each exception or error is related to a number. That number is used in the IDT to index the proper handler. For example, if the user executes a program that tries to read memory from an address that does not exist, the OS will index the IDT at 14 (0x0e). This index is where the page fault handler is located. The handler is called and intern handles the exception. When the handler is called the interrupt pushes the old EIP as well as the error code of the fault onto the stack. This is done before switching into the kernel mode and gives the page handler more information about the fault.

When an instruction accesses memory the process has to convert that virtual memory (the memory that the instruction accessed) e.g. 0x77FFDD00 to physical memory. This is done by looking up the Page Directory Entry (PDE) the PDE is an array of pointers to the Page Table Entry (PTE). When an application tries to write to the page referenced by the PTE with the privileges of a service or regular application, the application will page fault, if the page pointed to by the PTE has supervisor permissions set. This page fault will be handled by OverflowGuard’s page fault handler.

5.3.4 Control Registers (cr)/ Debug Registers (dr)

When an application has ring0 which is the highest level an application can get. That application can access/modify certain registers that aren’t available to applications in ring1+. These registers are as follows:

- cr0 - Contains system control flags, operating modes and state of the processor
- cr1 - Reserved
- cr2 - Contains the page fault linear address (where the page fault occurred).
- cr3 - Contains physical address of the base of the Page Directory Entry (PDE) table.
- cr4 - Contains a group of flags that enable architectural extensions, and indicate operating system or executive support for specific processor capabilities.
- dr4 & dr5 - reserved, previous processors alias these registers to debug registers dr6 and dr7 respectively.

²⁷ PaX Project (<http://pax.grsecurity.net>), 5/19/2004

- dr6 - reports the conditions that were in effect when a debug exception occurred.
- dr7 - contains the type of breakpoint that was hit

5.3.5 PaX/OverflowGuard Comparison

OverflowGuard is heavily based on PaX page protection. PaX page protection is thoroughly documented on PaX's website²⁸. PaX outlines how to prevent compromise of a system by marking the system's heap/stack/and other anonymous memory pages as either Supervisor or non-present. These flags effectively mark the pages as non-executable because code will not be able to write or read from these pages. Whenever an instruction execution or data access occurs on a page that is in supervisor mode or not present, it causes a fault. Next a simple compare is done, if the old EIP is equal to where the fault occurred it was an instruction execution on a page that should not be executable. If the EIP is not equal to where the fault occurred it is a data access, this routine will be referred to as execution verification. If it is a data access PaX set the PTE to user mode, then manually perform the memory walk logic and load the address the application requested into the DTLB which is the Data Translation Lookaside Buffer. PaX then resets permissions on the page and returns to the application. The application can then access the Data Translation Lookaside Buffer (the DTLB contains all data access cache where as the TLB a.k.a. ITLB contains all instruction fetch caches) which contains the address and not the page. Next time a data access occurs a walk is done to the PTE which has supervisor permissions set and PaX will perform this operation over again.

OverflowGuard has a different take on Pax Page protection. Instead of marking all pages supervisor, OverflowGuard initially marks them all as read-only. This is done to cut down on faults occurring, and saves overhead. For example if OverflowGuard marked all the pages as supervisor then any time an instruction situated in the image base's page which is suppose to execute code would page fault, and this would slow the system. Instead OverflowGuard will mark all pages as read-only any page that has a fault occur and is read-only will be modified and marked as supervisor mode. This effectively marks the heap and stacks as supervisor and leaves the executable images as read only. The detection and data access are basically the same as PaX.

5.3.6 Individual Analysis of Files

5.3.7 OGCenter.exe & OGConfig.exe

These are two files that OverflowGuard (OG) installs on the users system. The files act as a front end for configuring OG and for receiving notifications on overflow attempts. OGCenter.exe sits in the user's system tray and pops up when clicked on or if there was an overflow attempt. OGConfig.exe is executed through OGCenter.exe and offers the users all configuration options, like "Monitor Stack Only". OGConfig.exe also allows the user to perform tests such as stack overflow or heap overflow.

5.3.8 OGRebase.exe

As mentioned in previous sections, OverflowGuard (OG) uses rebases system files as an added protection method. However by default OG does not rebase system files. The user must enter the %windir%\system32\ and execute OGRebase. OGRebase takes in no arguments and performs all the rebasing. OGRebase does not use the ReBaseImage API mentioned in the previous section. Instead OG reads the PE Header and modifies the image

²⁸ PaX Project (<http://pax.grsecurity.net>), 7/6/04

base field in the Optional PE Header section. This method has many draw backs. First it appears that the OGREbase.exe, copies shell32.dll, kernel32.dll, ntdll.dll and msvcrt.dll into a different name. OGREbase.exe then manually rebases these DLLs. However, even on reboot the rebase does not affect the actual files and it is not certain if this tool does anything.

5.3.9 OverflowGuard.sys

OverflowGuard.sys is the meat and potatoes of this protection suite. OverflowGuard.sys is what offers the user the Unix like buffer overflow protection. This driver is what setups and enforces the PaX documented outline.

5.3.10 Setup Phase

The driver's first phase is to overwrite the Interrupt Descriptor Table's (IDT) handlers with its own handler. OverflowGuard does this by calling a function designed specifically for this operation. This function first calls sidt which is used to store the IDT. The IDT is stored in a local variable next the base of the IDT is added to the interrupt OverflowGuard wants to hook, in this case 0x0e and 0x01, then the handler number is multiplied by 8 and the base gets added to that to get the specific handlers address.

The math is as follows:

$$\text{idt_handler} = [\text{base_of_idt} + \text{interrupt_id} * 8]$$

OverflowGuard then converts the IDT handler to a physical memory address. This is done so that OverflowGuard can find the page that the IDT handler resides on. After OverflowGuard has the page that the interrupt handler is located on, OverflowGuard will set that page to supervisor mode. This is done to prevent malicious applications from overwriting OverflowGuard's interrupt handlers with their own.

Next OverflowGuard flushes any reference to the old interrupt handler by executing the invlpg (Invalidate Translation Lookaside Buffer) instruction. Finally the overwrite occurs, this is the most complicate operation of this phase. This is because the IDT entry is actually two long integers.

The entry looks like:

31 - 16	1 5	14- 13	12-8	7-5	4-0
Offset	P	DPL	0 D 1 1 0	0 0 0	Reserved

31 - 16	15-0
Segment Selector	Offset

To successfully hook the IDT OverflowGuard must overwrite the offset in both integers. The 31-16 bit offset must be the high word of the new interrupt handler address. The 15-0 must be the low word of the new interrupt address. This procedure is repeated twice once for each interrupt OverflowGuard overwrites.

Next OverflowGuard checks to see if the user has enabled ret-libc protection. If the user has indeed selected ret-libc protection, OverflowGuard will then setup four system-call hooks, which were explained in the previous section. The following functions are hooked:

- NtCreateSection
- NtCreateFile
- NtOpenFile
- NtProtectVirtualMemory

How OverflowGuard attempts to protect users will be explained later in this document.

5.3.11 Setting Memory Permissions

For OverflowGuard to be effective as a memory protection suite it must prevent the execution of malicious code. To do this OverflowGuard has to modify memory permissions, to cause a page fault so that OverflowGuard's page fault handler can be called. OverflowGuard's solution to setting the page permissions is relatively simple, OverflowGuard walks the PDE and every PTE is set to read-only. OverflowGuard also checks to make sure that the Global Page flag is not set. If it is set in the cr4 register OverflowGuard clears it. The Global Page flag allows frequently used memory pages to be shared globally and when the TLB is flushed the Global Pages are not, this effectively would bypass OverflowGuard and therefore is very necessary for OverflowGuard to success in protecting the user.

5.3.12 Catching New Processes

OverflowGuard offers limited protection in that it does not protect 3rd party applications. OverflowGuard is able to monitor new applications being executed by calling PsSetCreateThreadNotifyRoutineError: Reference source not found. When OverflowGuard's callback routine is called OverflowGuard will check the Create flag specified in the callback routine's parameters. If this flag is set to one then the thread is being created however if this flag is set to zero the thread is being closed (deleted). OverflowGuard handles both cases, first if the thread is being created OverflowGuard calls a function searches a linked list created by OverflowGuard that takes the thread ID as a parameter. This function's linked list routine looks up a table. The look up algorithm goes as follows:

```
Index_val = thread_id & 0x0FFF
```

The table looks something like this (we are calling it ProclInfo):

```
struct ProclInfo
{
..
..
/*0x0C*/    int ProtectStackValOption;
/*0x10*/    *TEB
/*0x1C*/    int ThreadID;
/*0x18*/    struct HashTable * pNext;
/*0x20*/    int ProcessID;
/*0x2C*/    *EPROCESS;
/*0x34*/    int ExitStatus;
/*0x124*/   char ProcessName[]
} //Total Size is 0x144
```

The linked list function indexes an array that we'll call ProcArray, this array is indexed at `index_val * 4`, in pseudo code this looks like:

```
ProcArray[ hash_val * 4 ]
```

If there is a value at the index then, OverflowGuard compares the thread ID at that index to the thread ID passed into the hash function, if the resulting compare is successful, and it is found that the thread is already in ProclInfo, then OverflowGuard returns a non-zero value. However if this index returns NULL, then OverflowGuard's linked list function returns zero.

It should be noted that the structure above is believed to be correct but, it is missing some member variables. Also this structure is used throughout the program as a way to store information about each process.

If the thread ID is found in the ProclInfo table's structures then OverflowGuard returns from its notify routine. Otherwise OverflowGuard continues, and calls PsLookupProcessByProcessId²⁹ which takes in a process ID and a pointer to the EPROCESS structure. OverflowGuard next indexes the EPROCESS's ImageFileName member located at offset 0x1fc. OverflowGuard compares the ImageFileName with a list of processes it protects. If this process is found, OverflowGuard will continue to fill out a new ProclInfo entry and point the pNext to equal the new structure.

However if the Creation flag is zero then the thread is being closed. OverflowGuard simply walks the linked list and removes any entry with the thread ID equal to the thread ID being closed.

5.3.13 Ret-Libc Protection

OverflowGuard claims to offer ret-libc protection. Ret-libc stands for return-into-libc which was discussed prominently by Solar Designer³⁰. The exploit demonstrated how to bypass non-executable stacks. This was done by overflowing the stack and instead of overwriting the return address to point to the attacker's shellcode, the return address would point to a libc function most likely system. Below the return address the attacker would place the parameter, usually a pointer to /bin/sh. Ret-libc exploits are not all that common on win32 platforms, they have not been explored in great detail. The Ins1der posted a ret-libc exploit for the RPC DCOM vulnerability found by LSD. This exploit as of 6/7/04 bypasses both OverflowGuard and StackDefender 1 and 2.

OverflowGuard attempts to protect against ret-libc exploits by hooking the functions previously mentioned in the Setup Phase 5.3.10. When one of the hooked functions is called OverflowGuard will first call KeGetCurrentThread³¹ which returns a pointer to a KTHREAD structure this structure is passed into a function that we'll call RetLibcVerify. RetLibcVerify indexes KTHREAD's UniqueThread member at offset 0x1e4. OverflowGuard then checks to see if the UniqueThread exists in its ProcArray. If the UniqueThread does not exist in the linked lists OverflowGuard executes the API normally. Otherwise OverflowGuard using the EPROCESS structure indexes the TEB (Thread Environment Block), then it adds the TEB to the ProclInfo structure and returns after verifying certain options, before RetLibcVerify returns it stores the ProclInfo structure that was identified in a global variable. Next OverflowGuard checks to see if an OBJECT_ATTRIBUTES structure was defined if one was not, then OverflowGuard calls the API and returns. Otherwise OverflowGuard calls a function which

²⁹ PsLookupProcessByProcessId (http://www.rootkit.com/newsread_print.php?newsid=139), 6/7/04

³⁰ Solar Designer Ret-Libc (<http://www.groar.org/expl/intermediate/ret-libc.txt>), 6/7/04

³¹ KeGetCurrentThread (http://www.osr.com/ddk/kmarch/k105_8fxu.htm), 6/7/04

checks the ObjectName in the OBJECT_ATTRIBUTES structure. This check, makes sure that the Object does not have certain privileges. Though in disassembly it was unclear which it checked for. The routine just described is used for NtCreateFile, NtOpenFile, NtCreateSection. NtProtectVirtualMemory has its own routine. This is described below.

The hooked NtProtectVirtualMemory works differently than the previously mentioned hooks. Instead of waiting to call the API NtProtectVirtualMemory gets called before any parameters are verified. OverflowGuard calls NtProtectVirtualMemory, OverflowGuard then calls KeGetCurrentThread and pass the KTHREAD structure into RetLibcVerify the same function that the other hooked API's call. This function is described previously therefore it will not be described again. OverflowGuard checks to make sure that the global variable that ProCInfo is stored in is not NULL. OverflowGuard then calls ExGetPreviousMode³² which returns the mode the processor is in this can be user or kernel mode. OverflowGuard then checks to see if the current thread is in kernel mode or user mode. If the thread is in kernel mode OverflowGuard will leave, however if the thread is not in kernel mode and is in user mode OverflowGuard continues to inspect the call. OverflowGuard next checks the 4th argument given to NtProtectVirtualMemory which is NewProtect. This argument specifies what permissions the page will have i.e. PAGE_READWRITE. OverflowGuard does a test on arg4 with the following binary sequence:

```
10101010
```

The only test scenarios where NtProtectVirtualMemory will not have its pages modified are cases where a test instruction will not result in one:

```
PAGE_NOACCESS
PAGE_GUARD
PAGE_NOCACHE
PAGE_EXECUTE_READ
```

If one of these flags is specified NtProtectVirtualMemory will return. However if a different flag is specified OverflowGuard will check to see if the thread calling NtProtectVirtualMemory is owned by a process that it protects. If OverflowGuard indeed owns the process OverflowGuard will verify that NtProtectVirtualMemory's ProtectSize parameter is greater than 0. OverflowGuard then marks each address that was just allocated as not present. This is done by looping from the base address of an allocated region and for every page that is found, the 1st bit in the PTE is changed to 0 to mark the page as not present.

5.3.14 Page Fault Exception

When a page fault occurs OverflowGuard will first check to make sure the page fault occurred in user-land and not in kernel space. If the fault was in user-land OverflowGuard will check to see if the page's permissions are set to read-only. This is done by translating the linear fault address to its PTE. Once this is done OverflowGuard checks the 1 bit in the PTE if it is zero then that page is read-only. If this is the case, OverflowGuard will set the page's permissions to user-level and remove the read-only flag. This is done by modifying the 2nd bit in the PTE and making it 1 and then changing the 1st bit in the PTE to 1 as well. OverflowGuard then manually performs the data walk, forcing the new address that caused the fault to be stored in the DTLB (Data Translation Lookaside Buffer), next OverflowGuard resets the permissions except this time, instead of the page being read-only it is marked as supervisor which will cause any access from anything other than the kernel to cause a page fault. OverflowGuard does this by changing the 2nd bit of the PTE to zero.

³² ExGetPreviousMode (http://www.osr.com/ddk/kmarch/k102_392q.htm), 6/7/04

When the page fault occurs OverflowGuard detects the page is not in supervisor mode OverflowGuard then performs some routine checks. OverflowGuard checks to make sure that the fault address is not equal to the old EIP, if this is the case then OverflowGuard will mark this as an attempt to execute code on a non executable page e.g. stack or heap. However if the fault is not equal to the old EIP then OverflowGuard decides that it was a data access and repeats the manual loading of the address into the DTLB as mentioned above.

5.3.15 Debug Exception

When OverflowGuard catches a Debug Exception, the debug exception handler will check if the page is properly aligned. Then OverflowGuard will check to see if the processor is in single step mode, this is usually a sign of a debugger attached to the system. OverflowGuard checks this by looking at the dr6 (debug register), the dr6 will contain information on the debug exception. If the 14th bit is set in dr6 the processor is in single step mode, OverflowGuard will effectively change this by setting the bit to 0 and then continue to examine the exception. OverflowGuard then goes through the memory of the thread that caused the violation and resets the permissions on those pages that are referenced by the thread. This is done to prevent a program from taking over the debugging procedures and bypassing the page fault routine.

5.3.16 Defeating OverflowGuard

OverflowGuard states on their website "In "Protect Only Selected Services" mode, which is enabled by default, OverflowGuard only protects services which have been tested to work properly with OverflowGuard. When "Protect Only Selected Services" is disabled all installed services are protected." OverflowGuard says to only protect services. Therefore they offer no protection to third party applications that do not run as services. This is a major problem because as the reader knows some of the most popular applications used by users are not services e.g. IM clients, file sharing, MP3 players etc. So when OverflowGuard was tested against the Attack Vector Test Platform it failed, the results below illustrate the ineffectiveness of protecting third party non service applications. These tests were preformed with ret-libc detection enabled and selected service protection turned off.

OverflowGuard Attack Vector Test Platform Results

A plus symbol (+) indicates that the software successfully protected against the specified exploitation vector.

Buffer overflow on stack all the way to the target

- Target: Parameter function pointer
- Target: Return address
- Target: Function pointer

Buffer overflow of pointer on stack and then pointing to target

- Target: Parameter function pointer
- Target: Return address
- Target: Function pointer

5.4 NGSEC StackDefender 2.0

5.4.1 StackDefender Overview

Note: Due to an un-expected update of NGSEC's StackDefender we were unable to get a full analysis done by the deadline. An updated section will be released at a later date.

NGSEC cut back considerably on the install files for StackDefender. In the 2.0 version, StackDefender does not install any *NG.fer files nor does it provide any user level API hooking.

Instead StackDefender 2.0 offers driver enforced buffer overflow protection. The method used by StackDefender 2.0 is very different from that used in 1.0. StackDefender 2.0 does not hook user level API functions such as CreateProcessA or VirtualAlloc. Instead StackDefender 2.0 acts more like a PaX based protection system. StackDefender 2.0 attempts to prevent the allocation of writeable and readable memory. StackDefender prevents the allocation of writeable and readable pages by hooking ZwAllocateVirtualMemory and attempts to prevent the conversion of pages from read-able to writeable and executable by hooking ZwProtectVirtualMemory. StackDefender also implements two IDT hooks, one IDT entry hooked is the page fault handler 0x0e, the other IDT entry hooked is syscall entry 0x2e. StackDefender works in a similar way to OverflowGuard in that it marks pages either read-only or supervisor. This forces the page fault and the page fault handler handles the violation. The syscall entry hook makes sure any process calling a system call has proper memory permissions set.

5.4.2 Bypass

StackDefender 2.0 pales in comparison to its previous version 1.10. StackDefender 2.0 doesn't catch any third party applications though it claims to. StackDefender 2.0 also has a significant CPU overhead that is noticeable after the first install. Just like OverflowGuard StackDefender 2.0 does not catch any third party applications. The Attack Vector Test Platform results are as follows:

StackDefender 2.0 Attack Vector Test Platform Results

A plus symbol (+) indicates that the software successfully protected against the specified exploitation vector.

Buffer overflow on stack all the way to the target

- Target: Parameter function pointer
- Target: Return address
- Target: Function pointer

Buffer overflow of pointer on stack and then pointing to target

- Target: Parameter function pointer
- Target: Return address
- Target: Function pointer

6 Attack Vector Test Platform Results

	PaX	StackGuard	StackShield	ProPolice SSP	Visual Studio .NET	OverflowGuard	StackDefender 1.10	StackDefender 2.0
Stack overflow to target								
Parameter function pointer	+	-	-	+	+	-	+	-
Parameter longjmp buffer	+	-	-	-	N/A	N/A	N/A	N/A
Return address	+	+	+	+	+	-	+	-
Old base pointer	+	+	+	+	N/A	N/A	N/A	N/A
Function pointer	+	-	-	+	+	-	+	-
Longjmp buffer	+	-	-	+	N/A	N/A	N/A	N/A
Heap/BSS overflow to target								
Function pointer	+	-	-	-	N/A	N/A	N/A	N/A
Longjmp buffer	+	-	-	-	N/A	N/A	N/A	N/A
Pointer on stack								
Parameter function pointer	+	-	-	+	+	-	+	-
Parameter longjmp buffer	+	-	-	+	N/A	N/A	N/A	N/A
Return address	+	-	+	+	+	-	+	-
Old base pointer	+	+	+	+	N/A	N/A	N/A	N/A
Function pointer	+	-	-	+	+	-	+	-
Longjmp buffer	+	-	-	+	N/A	N/A	N/A	N/A
Pointer on heap/BSS								
Return address	+	-	+	-	N/A	N/A	N/A	N/A
Old base pointer	+	+	+	+	N/A	N/A	N/A	N/A
Function pointer	+	-	-	-	N/A	N/A	N/A	N/A
Longjmp buffer	+	-	-	-	N/A	N/A	N/A	N/A

7 Conclusion

Many options are available for users seeking a solution to the widespread exploitation of buffer overflow vulnerabilities. The test results show that there are varying coverage capabilities between the different software titles analyzed, and our research has shown that attackers are still one step ahead with methods available to defeat almost every protection mechanism available. The combination of kernel and compiler based protection software is currently the best defense. Compiler protections are able to modify the structure of the generated binary itself and implement run-time checks, while the kernel is able to protect the environment in which the program runs and reduce the reliability of exploits which require hardcoded addresses. Although the currently available solutions may not be perfect, they are certain to help combat the proliferation of automated exploitation by worms and amateur attackers.