eEye Digital Security®

eEye Digital Security
Technical White Paper

Generic Anti-Exploitation Technology for Windows

For more information about eEye Digital Security, visit:
www.eeye.com

Table of Contents

Abstract

This paper will perform an impartial examination of generic anti-exploitation technology for the Windows platform. Beginning with a brief tour of the most important historical anti-exploitation projects, we will then analyse recently introduced security features in Windows XP, Service Pack 2 and Windows 2003, Service Pack 1, and summarise the remaining areas of vulnerability. Finally, we will discuss the various general approaches taken by 3rd party technology and also examine some possible future developments.

Scope

Although this paper will attempt a detailed examination of the available and prospective technologies that can provide generic anti-exploitation for Windows, it should not be inferred that such technologies can provide comprehensive security in and of themselves. In fact, based on the research presented, it should become clear that this breed of technological solutions is never likely to fulfill the promise to "make malware a thing of the past". Nevertheless, anti-exploitation technology is a useful, indeed critical, addition to a defense-in-depth security posture.

Readers are assumed to have a fairly sound understanding of stacks and heaps, passing familiarity with x86 assembler and CPU registers, and an understanding of basic programming concepts like pointers and functions. Some sections are technically heavy, and are intended to illuminate the referenced source material rather than to completely re-hash it, so further reading is highly recommended.

Definitions and Terminology

Many of the definitions in this area of security are not set in stone. It is not intended here to redefine any terms, but simply to explain the way they are used in this paper.

Referring to "anti-exploitation" technology is lamentably vague. When discussing anti-exploitation technology in this paper, we focus mainly on a specific class of attacks that use program input to cause memory corruption and subsequent code execution. These attacks may be delivered locally or remotely, via network input, file contents, environment variables, command-lines or many other means. The simplest of these attacks are stack or heap-based buffer overflows, but also included are integer errors (which are essentially delivery mechanisms for buffer overflows) and format string bugs; so referring to anti-exploitation technology as "buffer overflow protection" would not cover all of its aspects. Below, we will introduce "control-flow attacks" as a more general term for this class of attacks.

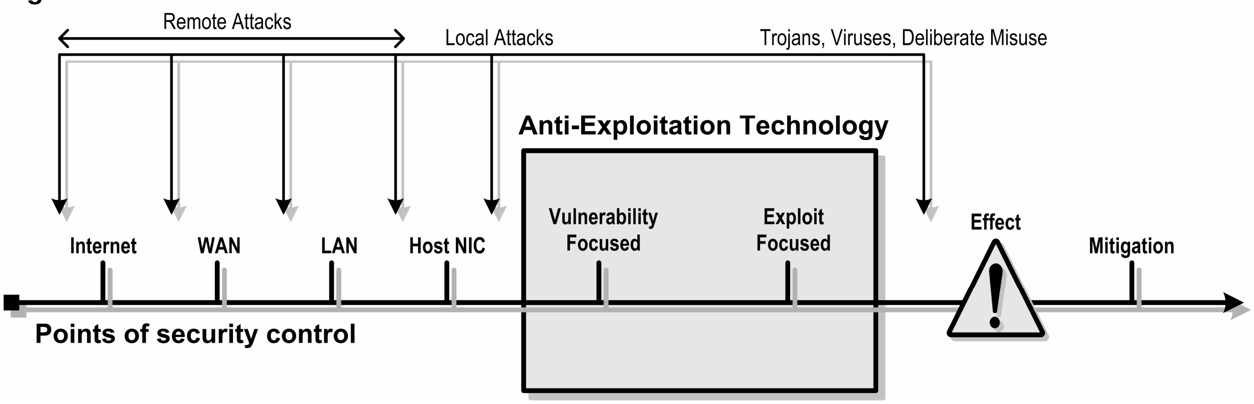
Vulnerabilities such as the ones discussed above are said to be "exploited" by attackers, which leads to the use of the word "exploit" to describe specific, attacker input. Exploits may be delivered via various means - for example a malicious .JPEG file may be an exploit, for certain vulnerabilities.

Finally, for those exploits that result in the execution of attacker-chosen code the exploit frequently includes "shellcode", which is a string of processor specific machine code, or "opcodes", that will be executed. Shellcode is sometimes also referred to as an exploit's "payload".

We have broken anti-exploitation technology into two types – measures that are "vulnerability focused" and those that are "exploit focused". As an example of our terminology, schemes that aim to detect and prevent buffer overflows at function return are "vulnerability focused" while those that aim to detect API calls from the stack are “exploit focused” since they address exploitation behaviour which could arise from many different kinds of vulnerability.

The figure below outlines the areas that will be discussed within this paper.

Figure 1: Points of Attack and Defense



Cause and Effect

Although self-evident, it must be noted that no technology can provide protection against attacks that bypass it completely. For example, in the figure above, network technologies like perimeter firewalls are unable to protect against attacks from the local segment. Similarly, anti exploitation technology is unable to protect against attacks which do not exploit one of a specific set of vulnerabilities. Attacks such as Trojan horses, phishing, viruses and similar threats requiring user interaction cannot usually be prevented by this technology. Pure data vulnerabilities, like the growing incidence of SQL injection attacks, for example, also bypass anti-exploitation technology, since the SQL server is behaving normally and has simply been passed an undesirable query by a script. In other words, even if a perfect solution to the problem of vulnerability exploitation were to exist, there are many classes of attack which it cannot address.

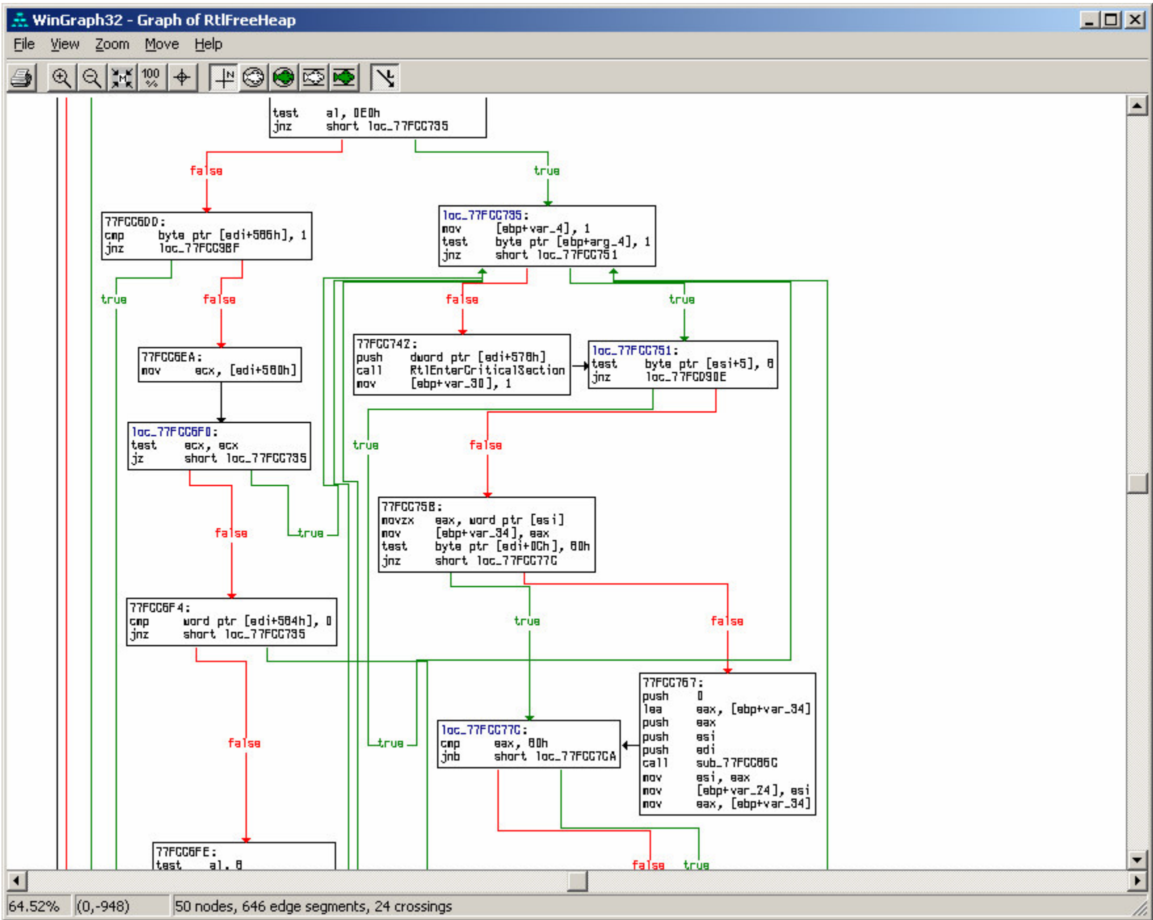
Program Control Flow

As a prelude to discussing the various vulnerabilities and protection strategies, it is useful to examine the concept of a control flow graph (CFG). Considering both vulnerabilities and protection strategies in terms of their effect on program control flow allows for meaningful comparisons and clearer understanding of the concepts involved.

A control flow graph is a graph that represents all possible paths of code execution within a program. Each block of contiguous code within the program forms a “node” in the graph, whereas all control transfers like JMP, CALL and RET represent edges – in other words connections between the nodes. This concept is important in compiler design, and more information is available in materials related to that field – although a brief summary can also be found in Wikipedia[1].

1 Wikipedia entry, *Control Flow Graph*, 2005, http://en.wikipedia.org/wiki/Control_flow_graph, accessed June 20, 2005

Looking at a code graph generated by IDA Pro[2], the nodes and edges can be easily seen:



A very important concept to recall is that this control flow graph should never change for the huge majority of programs – because the graph includes all possible branches. The exceptions are those programs that need to perform self-modification of code - such as the runtime code generation used by some interpreters, or certain device drivers that require extreme performance. This is why code segments within an executable file are typically marked as read-only (and also why data segments are typically marked as non-executable, but more on that later).

In general, we can classify the effect of an attack by examining what effect it has on the control flow graph. Stack smashing attacks, for example, are “code injection” attacks, since they inject new code (adding a node) and then modify a pointer in memory (adding an edge) to redirect execution to the payload. Other, more subtle exploits for buffer overflows use pointer corruption without the need to inject code – merely adding or shifting edges in the graph. Probably the most famous example of this class are “return-into-libc” attacks, where fake function parameters are pushed onto the stack and control is transferred to a library function. The canonical example is to push “/bin/sh” and modify the function’s return address to point to system().

[2] IDA Pro is a widely used disassembler and debugger. More information at http://www.datarescue.com/

The most subtle attacks, "pure data" attacks, corrupt program data but leave the CFG completely untouched – for example overwriting a security critical data structure which is later used in the program logic. This general classification of exploits and vulnerabilities will be used throughout this paper. Any attack which relies on modification of the control flow of a program (buffer overflows, pointer attacks, integer errors and format string vulnerabilities all fall into this category) will be referred to as a control-flow attack.

History of Anti-Exploitation

In order to set the scene for an examination of Windows anti-exploitation technology, it is useful to quickly summarise the key historical developments which have shaped this field. It is not intended, however, to provide a truly comprehensive examination of these tools. Where possible we have provided relevant references, which can be consulted for more detailed descriptions. However, understanding the basic concepts outlined will greatly simplify the discussion of the design and implementation of the current generic technology for Windows.

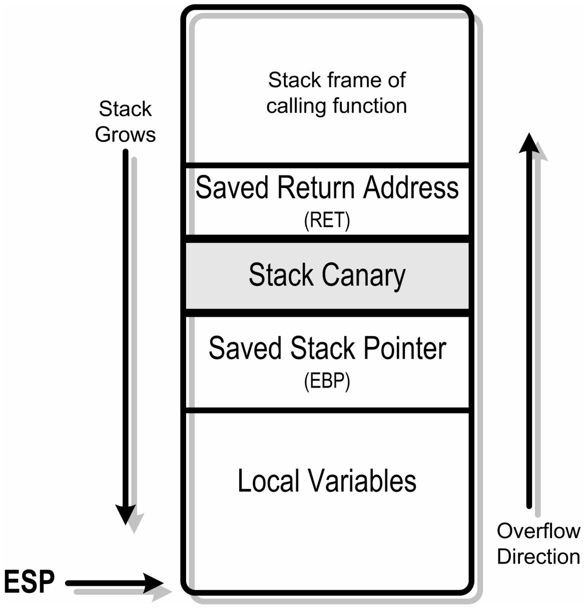
Most significant developments in the field have been under UNIX. In that realm, there are three main ways in which protection can be added – compiled-in, linked or injected and via kernel modifications.

Compiled-In

Compiled-in systems use compiler modifications to create program binaries that are more resilient to control flow attacks. The best known among these systems is Crispin Cowan's StackGuard[3], which was also used as the basis for the Microsoft /GS compiler option, discussed in much more depth below. Other systems along these lines include ProPolice / SSP[4], from IBM Research and StackShield[5]. Each of these three systems takes a different basic approach, but all result in constructing stack frames that attempt to detect buffer overflows at the point of compromise (as the function returns). In our terminology these are vulnerability focused approaches. Simplified, the three approaches are as follows:

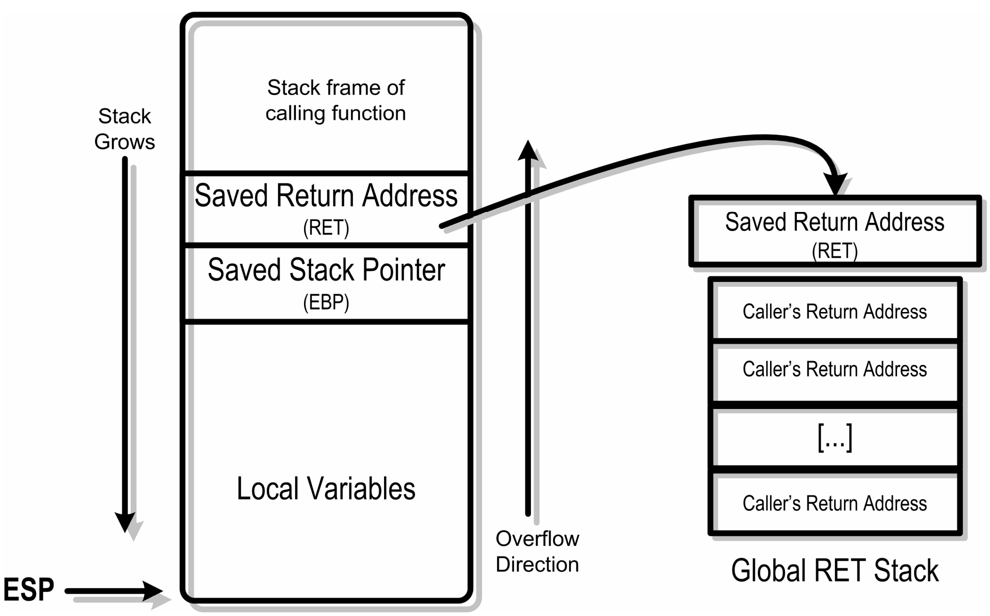
StackGuard coined the term "stack canary", for a value that is placed in front of the saved return address. The concept is simple – any overflow will trample the canary before it is able to modify the return address. When the function returns, the canary can be checked against the stored value – a mismatch indicates that something has gone wrong.

[3] C. Cowan et al. *StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks*. In USENIX Security Conference, January 1998. online at http://sherry.ifi.unizh.ch/cowan98stackguard.html
[4] H Etoh, *GCC extension for protecting applications from stack-smashing attacks*, http://www.research.ibm.com/trl/projects/security/ssp/, June 20, 2005
[5] Vendicator, Stack Shield, http://www.angelfire.com/sk/stackshield/, June 20, 2005



As with many things, however, the devil lies in the details. We will discuss some implementation limitations in this concept when examining the Microsoft /GS compiler option. StackGuard is one of the projects that can be considered to have significantly influenced the direction that Microsoft themselves have taken with the built-in anti-exploitation features in Windows 2003 and Windows XP, Service Pack 2.

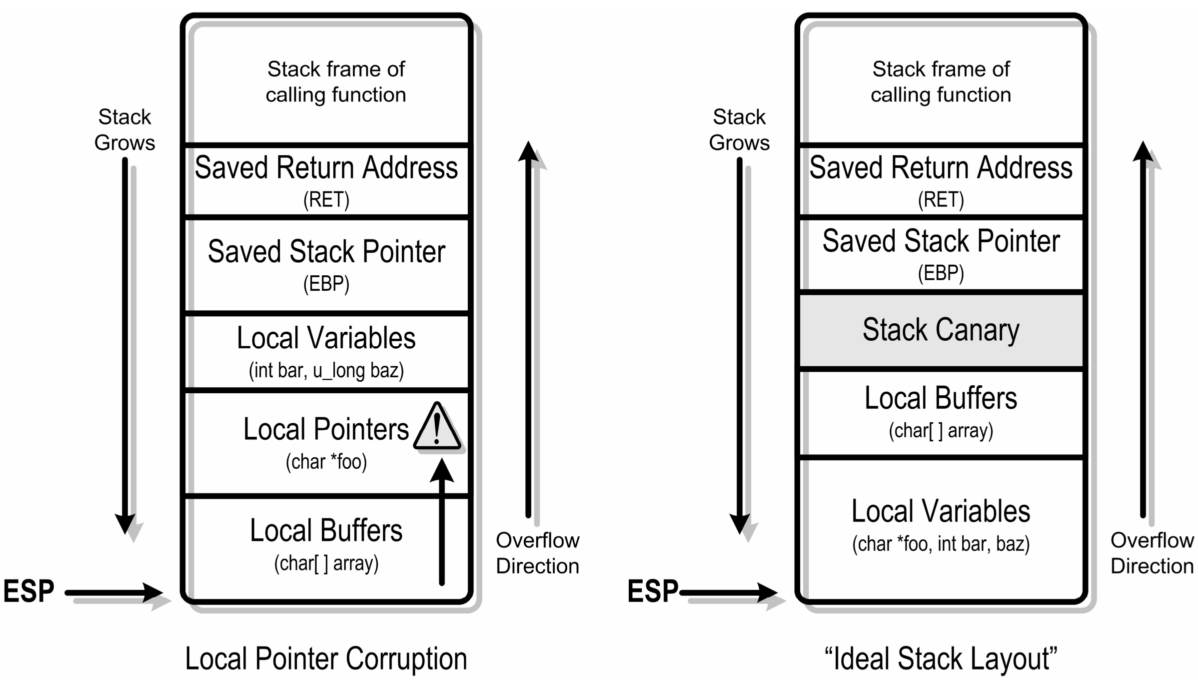
StackShield takes a different approach, using a “global ret stack”, which is really just another way of protecting the return address. As each new stack frame is added, the correct return address is pushed onto a “ret stack”, which is stored on the heap. Under Stack Shield, the process stack layout remains the same – the checking is all done in the function epilog.



While StackShield implements the same basic concept as StackGuard (and the attacks against them often appear in the same papers) it contains some other interesting ideas. "Return range checking" will prevent attempts to return to an address higher than a given base – intended to stop returns into the stack and heap. Ranges are also checked when a call is made based on a function pointer (indirect calls) – designed to prevent control flow hijacking by overwriting function pointers, which is a very common technique. These checks are not perfect; Gerardo Richarte from CORE points out some important implementation errors with StackShield's range checking in "Four different tricks to bypass StackShield and StackGuard protection"[6]. However, although it was implemented poorly in StackShield, the generic approach of checking the validity of caller and return addresses is used extensively in 3rd party Windows anti-exploitation solutions.

The final compiled-in solution examined is ProPolice / SSP, from IBM Research. Propolice also uses a stack canary, but adds a refinement also adopted by later versions of StackGuard and the Microsoft /GS option – the canary also protects the saved stack pointer (EBP). This sensible modification makes several of the public attacks against StackGuard and StackShield infeasible – some specific EBP attacks such as the off-by-one technique are discussed in Phrack 55, 0x08[7] as well as in Richarte[8].

The real innovation in ProPolice, though, is the "ideal stack layout". The idea here is to re-order variables on the stack, such that the resultant damage from an overflow is minimized – pointers are stored below overflow-prone buffers, which are placed directly before the stack guard value. In this way a buffer overflow is unable to modify a stored function pointer. The layout differences are shown below.



[6] G Richarte, *Four different tricks to bypass StackShield and StackGuard protection*, 2002, http://www1.corest.com/files/files/11/StackguardPaper.pdf
[7] Klog, *Frame Pointer Overwrite*, 1999, http://www.phrack.org/show.php?p=55&a=8
[8] Above, note 6

It should be mentioned, however, that due to an implementation error very small arrays will not be correctly re-ordered.

Compiled-In Solutions – Summary

Unfortunately, these systems leave very important areas of exploitation untouched. None of them provide any protection for heap memory (probably the biggest flaw), and all of them benefit greatly from the additional protection of a non-executable stack (discussed further below). In fact, when Wilander and Kamkar tested all of the solutions above against a variety of control-flow attacks in 2003 they noted that that "none of these can handle the diverse forms of attacks known today. In practice at best 40% of the attack forms were prevented and another 10% detected and halted, leaving 50% of the attacks still at large."[9]

However, these seminal solutions have influenced some of Microsoft's own direction in the anti-exploitation field. Third party solutions, however, are unable to take advantage of compiled in protection for the huge majority of Windows executables, so those solutions have evolved along very different lines.

Linked / Injected

Baratloo, Tsai, and Singh introduced a novel approach in 2000 with libsafe and libverify[10]. Libsafe is a dynamically loaded library which replaces "unsafe" function calls such as strcpy(), strcat() and sprintf() with safer equivalents. The original functions are 'hooked', and a new version of the function – including check code – is executed instead. Function hooking is a technique which is encountered frequently in the Windows anti-exploitation world, although we were unable to identify any Windows implementations which replace dangerous calls with "safe" replacements.

Libverify is a generic stack protection approach which uses runtime code injection to hook every userland function with an entry and exit wrapper. Although the authors describe the solution as "similar to StackGuard" it is actually more similar to Stack Shield – the prolog pushes the correct return value onto a "ret stack" which is checked on return. Also, the canary values are actually the correct return addresses, unlike StackGuard's terminator or random cookies.

The most interesting thing about the libverify design is that it does not require recompilation. As part of the dynamic linking process, the library's _init() code is run, which installs the hooks. A similar approach is possible under Windows, with Dynamic Link Libraries (DLLs) and the DLLMain() function. This approach is much more suitable for a third party extension to Windows than the compiled in options discussed above, and has been adopted and extended by several third party vendors. Simple injection via DLLMain() is prone to a race attack in some cases, so some systems use alternate techniques such as hooking the process creation routines in the kernel.

This method of implementation was innovative, with its ability to protect existing systems without recompilation, and the performance of libsafe in particular is extremely impressive (in some cases, libsafe

[9] J. Wilander, M. Kamkar. *A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention*. In Proceedings of the 10th Network and Distributed System Security Symposium, 2003, online at http://www.ida.liu.se/~johwi/research_publications/paper_ndss2003_john_wilander.pdf
[10] A. Baratloo, T. Tsai, and N. Singh. *Transparent Run-Time Defense Against Stack Smashing Attacks*. In Proceedings of the USENIX Annual Technical Conference, 2000, online at http://www.research.avayalabs.com/project/libsafe/doc/usenix00/paper.html

functions outperformed their “dangerous” equivalents). Unfortunately, the security performance was not notably better than the stack protection systems discussed above. Security testing results can be seen in Wilander[11].

Kernel

No discussion of anti-exploitation technology would be complete without an examination of PaX. It might be also argued that no discussion of UNIX anti-exploitation is complete without a reference to the vitriolic mailing list exchanges between the proponents of ProPolice vs StackGuard, PaX vs OpenBSD's W^X and all things in between. Virtually any bugtraq search on those keywords around 2003 will turn up a litany of highly amusing threads – a cogent reminder that anything written to a mailing list remains in the public domain forever.

The PaX project[12] aims to apply many low-level changes to UNIX, most of which are far too complicated to explain in detail in this paper. The PaX documentation, however, is excellent, and should be required reading for any prospective developer of anti-exploitation technology – even on Windows. In fact, there are several Windows projects which use “PaX technology” to a greater or lesser degree (and with variable success).

Referring to the “where things fit in” diagram (figure 1) PaX is a set of “exploit focused” features, rather than a “vulnerability focused” approach, which is why it is normally combined with one of the anti-stack-smashing approaches discussed above. For example, Hardened Gentoo[13] uses PaX in combination with SSP, formerly known as ProPolice (discussed above).

The key concepts implemented in PaX are below.

Address Space Layout Randomization (ASLR)

PaX allows virtually every part of the process address space to be randomized. This feature is designed to complicate exploitation by making the attacker “guess” the program control flow, and thus make attacks harder to write.

In a basic buffer overflow attack, the attacker has two goals – firstly code injection (a new node in the Control Flow Graph) followed by a control transfer to redirect execution into the malicious code (a new edge). A quick review of introductory buffer overflow material will show that selecting an appropriate return address poses the greatest challenge, and is the most difficult technique for novice researchers to master. In simple terms, once the attacker is able to control the flow of execution, they need to decide to *where* it should be redirected. A mistake will almost certainly cause a program crash, which is useless to the attacker.

Many approaches have been created to simplify this process, but the simplest is the “trampoline”. Trampoline attacks involve accessing the address of the payload via a register, even if the absolute address will not be known in advance, and then finding a sequence of instructions in a fixed location, such as a library, that will execute a relative call such as “CALL [ESP+20]”. The key to the trampoline attack is that the attacker can now use *relative* addressing to locate the payload, instead of absolute addressing.

[11] Above, note 9
[12] *Homepage of the PaX Team*, http://pax.grsecurity.net/
[13] *Hardened Gentoo*, http://www.gentoo.org/proj/en/hardened/

Other exploitation techniques involve overwriting function pointers in predictable locations, like the Unhandled Exception Filter in Windows, or entries in the destructors (.dtors) segment in linux. Unlike trampoline attacks, these techniques are primarily designed to keep control of execution in cases where vulnerable functions do not return cleanly after the exploit. In such cases the vulnerable application is probably in the process of crashing – for example if the entire stack or heap has been corrupted, or if a stack canary has been trampled.

Both of these techniques are complicated by ASLR. With an address space that has been obfuscated by ASLR, the attacker doesn't really know the address of *anything*, which means that they can no longer use well-known addresses and offsets to redirect execution. ASLR is applied to the general layout of the process, randomizing the base locations of segments, libraries, as well as the stack and heap.

Attacking ASLR is possible, but an information gathering phase is required. A good outline of this can be found in Phrack 59, 0x09[14]. In summary, the author demonstrates the use of a format string bug to gather information about the target stack, allowing the address of libc to be determined – this is followed by a standard return-into-libc attack.

Another form of information leakage attack against ASLR was described in 2004[15]. This attack uses a blind "Oracle" brute force technique. Attacking a theoretical Apache stack overflow, the researchers guessed the base address of libc by using a ret-libc attack to attempt to call usleep(). When the base address was guessed incorrectly, the thread would crash and be respawned, but when the address of libc was guessed correctly the process would sleep – an information leak which was detectable remotely. Once the libc base address was known, a standard return-into-libc attack was possible.

However, attacks against ASLR which cannot exploit some kind of information leakage vulnerability are unlikely to be successful. The first attack above requires a format string vulnerability, and the second is a multi-shot exploit, which requires a multi-threaded server.

A very minor form of randomization has been introduced by Microsoft in their own early anti-exploitation efforts, with the PEB (Process Environment Block) base location being slightly randomized on XPSP2. This is discussed further below. Some 3rd party solutions attempt to implement ASLR on Windows, but Windows binaries are more sensitive to relocation (in particular some modules cannot be relocated), so these approaches are fraught with compatibility issues – this is also discussed further below.

PAGEEXEC / SEGMEXEC

The other key feature of PaX is to provide non-executable memory pages. For IA-32, PaX takes one of two approaches. It can provide an emulated NX (Non eXecutable) bit using PAGEEXEC at a performance penalty, or provide a solution with very low performance overhead using SEGMEXEC, but at the cost of halving the userland virtual address space (1.5GB instead of 3GB)[16]. Before discussing the implementation details, we should briefly cover the overall goal of NX memory pages.

---

[14] T. Durden, *Bypassing PaX ASLR protection,* Phrack 59, Article 9, 2002, available at http://www.phrack.org/show.php?p=59&a=9

[15] H Shacham, M Page et al, *On the effectiveness of address-space randomization,* in Proceedings of the 11th ACM conference on Computer and communications security, 2004 online at http://www.stanford.edu/~blp/papers/asrandom.pdf

[16] User space in 32-bit linux is 3GB, unlike 32-bit Windows which divides the 4GB of addressable memory into two equal halves by default. This can actually be modified using the /3GB boot.ini switch.

To return again to Control Flow Graphs, recall that most simple control-flow attacks rely on code-injection, followed by a control diversion. Injected code will always be in writeable memory – attempts to inject code directly into, for example, the read-only .text segment will result in an access violation, and usually a crash. Once the code is injected, the next goal must be to redirect execution such that the injected code will be run. Therefore, the key to NX protection is to disallow the execution of code from writeable memory – in other words, any data that is in a writeable page should never be interpreted and run as code. This is well summarized by the OpenBSD technology equivalent W^X (W xor X) – You can Write to it, or eXecute it, but not both.

This approach is far from new. As early as 1997, experts such as Casper Dik and Solar Designer had created patches to implement NX stacks on major UNIX operating systems[17],[18]. PaX extends this approach to cover the stack as well as the process heap; indeed potentially any page can be marked NX. Heap vulnerabilities were once considered to be unexploitable, even by experts – however they are now the subject of "cookbook" exploitation recipes, so extending NX protection to the heap is essential.

On IA-32 (normal Intel x86 processors), there is no direct CPU support for NX memory pages. On other architectures such as SPARC, Alpha and PowerPC the NX bit is supported in hardware. So, on processors that provide hardware support for NX, PaX will use the NX bit via the PAGEEXEC mechanism, at zero performance cost. For IA-32 things are more difficult.

The technical details of PAGEEXEC for IA-32 are beyond the scope of this paper, and are discussed in the PaX documentation[19]. In summary, it marks NX pages as not-present at the hardware page-table level. When those pages are accessed a page-fault will always be generated. PAGEEXEC code then handles the fault and checks to make sure that EIP is not within the same range – which of course would mean that execution had been transferred to an NX page. Assuming that is not the case, the memory access is then allowed. PAGEEXEC's performance is variable, because it depends heavily on how often these NX-related page faults are raised.

SEGMEXEC is actually a brilliantly simple concept, although the official PaX documentation is somewhat daunting. Essentially, SEGMEXEC splits the 3GB of addressable user-space memory into two halves. The bottom half is just as normal, but every memory page which is executable is copied into the top half – resulting in a perfect mirror copy with all of the data pages snipped out (such as the stack, heap, .bss etc). Then, there is a "trick"...

Intel protected mode architecture uses "selectors" to translate offsets into linear memory addresses. In other words, when you access address 0x1000 in the code segment (EIP is always assumed to access the code segment) you are really accessing [base address from CS selector] +0x1000. This is extremely well explained in the IA-32 protected mode architecture documentation[20]. The SEGMEXEC "trick" is to modify the CS selector so that the base is at 0x6000000 (1.5GB) instead of 0x0. Now, *whatever* happens to EIP it is "trapped" in the top half of addressable memory (ie [0x6000000]+some offset), where only the code pages are mapped. Another way to think about it is that the code can be still be injected, but EIP *can never be redirected to the hostile code*. The worst that can happen is for EIP to point to the upper mirror of the payload address, which will be empty and result in a segmentation fault.

[17] Solar Designer, *StackPatch*, http://www.openwall.com/linux
[18] C. Dik, posting to comp.security.unix, January 2, 1997
[19] PaX Team, PAGEEXEC Documentation, http://pax.grsecurity.net/docs/pageexec.txt
[20] Intel Corporation, *Overview of the Protected Mode Operations of the Intel Architecture,* http://www.intel.com/design/intarch/papers/exc_ia.htm

SEGMEXEC under linux operates at very low performance overhead, due to the fact that the core mmap() memory allocation routines have been re-written. This is obviously not possible under Windows, but the general technique could still be viable with some modifications.

Sadly, the PaX project officially terminated on April 1, 2005 due to a critical bug in the core enabling technology of the SEGMEXEC feature[21]. The project has been transferred to the GrSecurity team, and at the time of writing it is not yet known what effect this will have on PaX's long term development.

Other Projects

Interested readers are also invited to investigate a few other open source anti-exploitation projects which were never adopted, adopted and abandoned or simply not completed.

Oded Horowitz proposed a fascinating compiler modification called "Big Loop Integer Protection" in Phrack 60, 0x09[22]. Integer "overflows" are really just delivery mechanisms for stack or heap based overflows. The crux of his approach was to recognize that Integer vulnerabilities rely on the fact that the integer is a counter, and working to detect counters that suddenly became much too large.

The OpenBSD catch cry is "secure by default", and several of the technologies and approaches adopted by OpenBSD repay further study; in particular systrace[23], written by Niels Provos. Systrace is the first system mentioned in this paper which is based on behavioural enforcement (also casually referred to as "sandboxing"), which will be discussed in much more depth later.

In 2003, Roberston, Kreugel et al proposed a GNU libc modification to Doug Lea's malloc aimed at preventing heap overflows[24]. Much like StackGuard and similar systems, it proposed a heap "cookie". For some reason, no major operating systems seem to have real-world implementations of this technique – except Windows.

Crispin Cowan, of StackGuard fame, also proposed several other interesting ideas, among them "FormatGuard"[25], a C Pre-Processor macro which "defanged" dangerous format strings, and "PointGuard"[26], which aimed to protect all pointers at runtime.

PointGuard is a particularly important milestone project. To state the obvious, *all* control-flow attacks must at some point divert the control flow. This is virtually always done by attacking a pointer – either a function pointer, a saved return address, a relative linear address, saved longjmp buffer, exception handler or any of a myriad of methods. PointGuard ambitiously set out to protect most pointers while they were in

[21] Pageexec, *PaX privilege elevation security bug*, posting to bugtraq mailing list, March 5, 2005, http://www.securityfocus.com/archive/1/392348/2005-03-05/2005-03-11/0
[22] O. Horovitz, *Big Loop Integer Protection,* 2002, Phrack 60 article 9, http://www.phrack.org/show.php?p=60&a=9
[23] N. Provos, *Systrace - Interactive Policy Generation for System Calls,* 2003, homepage at http://www.citi.umich.edu/u/provos/systrace/
[24] W. Roberston, C Kruegel et al, *Run-time Detection of Heap-based Overflows*, 2003, in proceedings of 17th Large Installation Systems Administration Conference (LISA), online at http://www.usenix.org/publications/library/proceedings/lisa03/tech/full_papers/robertson/robertson.pdf
[25] C. Cowan, M. Barringer et al, *FormatGuard: Automatic Protection From printf Format String Vulnerabilities*, 2001, in proceedings of 10th USENIX Security Symposium, online at http://www.usenix.org/events/sec01/full_papers/cowanbarringer/cowanbarringer.pdf
[26] C. Cowan, S. Beattie et al, *PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities*, 2003, in proceedings of 12th USENIX Security Symposium, online at http://www.usenix.org/events/sec03/tech/full_papers/cowan/cowan.pdf

memory (pointers are safe while in CPU registers) using an encryption routine. In effect, PointGuard was designed to enforce Control Flow, although without trying to model it or "understand" it.

Having examined the key historical approaches, we can now examine the new anti-exploitation technologies recently introduced by Microsoft.

Windows Anti-Exploitation Technologies

In this section we will examine the anti-exploitation features recently added to Windows by Microsoft. It is worth noting that most of these technologies are available only on Windows XP, Service Pack 2 (XPSP2) and Windows 2003, Service Pack 1 (W2K3). The exceptions are the /GS and /SAFESEH compiler options, discussed below, which can protect any application compiled with Visual Studio .NET 2003 – but the Windows binaries from older operating systems have not been compiled this way, so only third party applications would be protected.

Here is a quick summary of the new technologies:

Protection	Applies	Focus
/GS Compiler Option (stack cookies)	Per App	Detect Attack
Compiler Stack Layout Optimization	Per App	Complicate Exploitation
Heap Cookies	Global	Detect Attack
Safe Unlinking	Global	Detect Attack
PEB Randomisation (XPSP2)	Global	Complicate Exploitation
Remove Pointers in PEB (2K3)	Global	Complicate Exploitation
Pointer Encoding, UEF (2K3), VEH	Global	Complicate Exploitation
NX (Hardware DEP)	Configurable	Detect Attack
Improved SEH security	Global / Per App	Complicate Exploitation

Protecting the Stack

The key technology available for protecting the stack in Windows is the /GS compiler option in Visual Studio .NET 2003. Essentially, /GS is very similar to StackGuard, with the minor modification that the security cookie also protects the saved frame pointer on the stack (usually referred to as EBP). Additionally, however, VS.NET 2003 also performs some optimization of the variables on the stack, in a similar manner to ProPolice / SSP – however the efficacy of these layout optimizations were questioned at CanSecWest in May, 2005[27] by the principal author of ProPolice.

One particular modification which was made between VS.NET versions 7.0 and 7.1 was to copy the function pointers to local exception handlers below local variables – recall that a buffer overflow runs "upwards" – making it impossible to overwrite these pointers with a classic buffer overflow. While it is possible that this was done in response to attacks such as those described by Cigital[28], Brandon Bray

[27] H. Etoh, *Stack Protection Systems: (propolice, StackGuard, XP SP2),* 2005, Presented at CanSecWest 2005, online at http://cansecwest.com/core05/propolice-cansec2005.ppt
[28] C. Ren, M. Weber, G. McGraw, *Microsoft Compiler Flaw Technical Note*, 2002, online at http://www.cigital.com/news/mscompiler-tech.pdf

from the Microsoft Visual Studio Team suggests an alternate explanation in his blog[29] "The VC 2003 release had a short development cycle, so not all of our ideas to improve /GS were implemented. The Whidbey product cycle gave us the opportunity to do more."

David Litchfield, however, pointed out several other approaches to bypassing this generic stack protection in Windows 2003 SP0[30]. These attacks brought into the spotlight some legacy implementation issues which undermined the security effect of the /GS compiler modification.

1. EXCEPTION_REGISTRATION structures, used by Structured Exception Handling are stored on the stack. Although the exception handler for the *current* call-frame is stored below any arrays and is safe from buffer overflows, the handler for the *calling* function (and any others) remains vulnerable. Overwriting pointers to exception handlers and provoking an exception (or just waiting for one to occur) is a well known attack pattern.
2. Microsoft's Safe Structured Exception Handling (SafeSEH) contained some baffling anomalies – for example allowing control to be dispatched to unregistered exception handlers on the heap. (discussed further below)
3. The Security Cookie used in the stack protection is stored in the .data segment, and not marked as read-only – sometimes allowing an attacker to overwrite it with a known value.

The key issues with the exception handling process have since been resolved, and much improved exception handling code is shipping in Windows 2003, SP1 and Windows XP, SP2, discussed further under SEH Security, below.

Although there are many application specific flaws which may allow stack-based overflows to be successfully exploited, previously known generic attack patterns are no longer viable against XPSP2 and Windows 2003 SP1

Protecting the Heap

Microsoft recently introduced two new vulnerability focused technologies to protect the heap. Heap overflows are a vulnerability area that is receiving more attention in recent years – the annual number of CVE entries for heap overflows tripled between 2002 – 2004[31], and heap exploitation is now considered to be reliable across Windows 2000 and Windows XP pre-SP2[32].

The general operation of heaps is beyond the scope of this paper, but some good general references (focused on exploitation) are "Vudo"[33] "Once Upon a Free"[34] and the indispensable Conover / Horovitz presentation from CanSecWest 2004[35]. The most important concept to understand is that pointers to free blocks of heap memory are kept "on the shelf" when they are not allocated – this is known as a freelist.

---

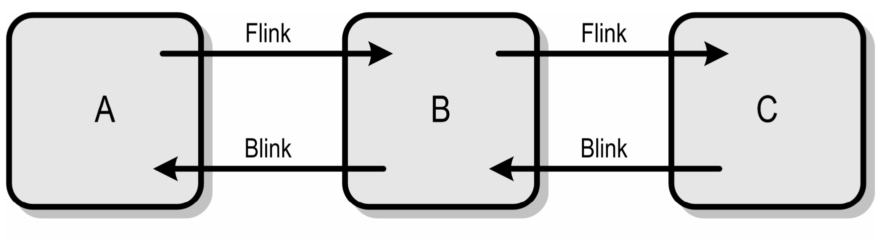
29 B. Bray, *Security Improvements to the Whidbey Compiler*, 2003, in weblog 'C++ Potential', http://blogs.msdn.com/branbray/archive/2003/11/11/51012.aspx
30 D. Litchfield, *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server*, 2003, http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf
31 MITRE, *CVE - Common Vulnerabilities and Exposures*, http://cve.mitre.org/ (statistic based on the author's coarse analysis of the CVE database, n=18 in 2002, n=53 in 2004)
32 Conover, below, note 35
33 M. Kaempf, *Vudo - An object superstitiously believed to embody magical powers*, 2001, Phrack 57, article 8, http://www.phrack.org/show.php?p=57&a=8
34 Anon, *Once upon a free()*, 2001, Phrack 57 article 9, http://www.phrack.org/show.php?p=57&a=9
35 M. Conover, O. Horovitz, *Reliable Windows Heap Exploits*, 2004, presented at CanSecWest 2004, online at http://cansecwest.com/csw04/csw04-Oded+Connover.ppt

When HeapAlloc() is called, Windows will check to see if it can recycle a block from a freelist before attempting to use fresh, unallocated memory.

To return to Windows, the two new heap protection technologies are “safe unlinking” and “heap cookies”.

Heap Cookies are an idea that seems to have first been formally proposed in 2003, in independent papers from Huang[36] and Robertson, Kreugel et al[37] although the fine details of the two proposed implementations vary. Certainly Microsoft seems to be the first vendor that has implemented them in production. The basic idea is almost identical to stack cookies – a ‘guard’ value is placed in the heap header, and is checked during heap operations which may be dangerous (for example during allocation), to ensure that corrupted blocks are not being allocated. One implementation limitation, however, is that the heap cookie is only 1 byte, and is therefore susceptible to brute force.

The other improvement in XPSP2 and W2K3 is “safe unlinking”, which is essentially a small sanity check. In a doubly linked freelist, the blocks are linked by forward and backward pointers, like this:

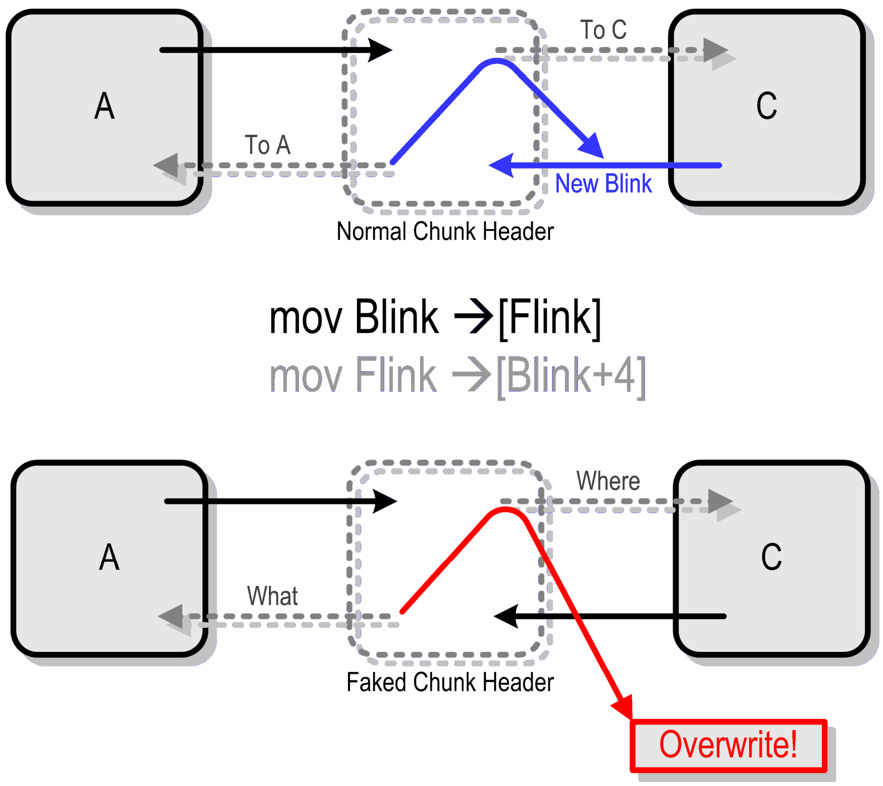


Doubly-Linked Free List

The forward and backwards links are normally known as Flink and Blink.

Safe unlinking is performed to prevent a well known vulnerability pattern which allows for an arbitrary memory overwrite. When Block B is unlinked, the Flink and Blink pointers for A and C need to be updated. Although this is slightly simplified, imagine that Windows then tries to update Flink(A) by copying the *value* of Flink(B) to the *location* pointed to by Blink(B) – which was intended to copy “pointer to C” over to Flink(A). By corrupting these pointers, an attacker that controls Block B can now write a chosen 32-bit value to any 32-bit location – this is known as the “4-byte overwrite”.

[36] Y Huang, *Protection Against Exploitation of Stack and Heap Overflows*, 2003, online at http://www.cgisecurity.com/lib/AntiOverflows.pdf
[37] Above, note 24



The safe unlinking check is to verify that the Blink of the block pointed to by Flink is correct, and the same for the Flink of the block pointed to by Blink. In other words, if Flink and Blink have been corrupted, when the links are "followed" there will be no corresponding return pointers. To summarise that, Conover / Horovitz[38] put it like this:

B→Flink→Blink == B→Blink→Flink == B (Block to be unlinked)

Conover, aka Shok, later described a technique at SyScan in December 2004 called "Unsafe Unlinking", which allowed the described check to be bypassed in a few rare cases[39], as well as outlining the remaining attack vectors against heap overflows on Windows XP, SP2.

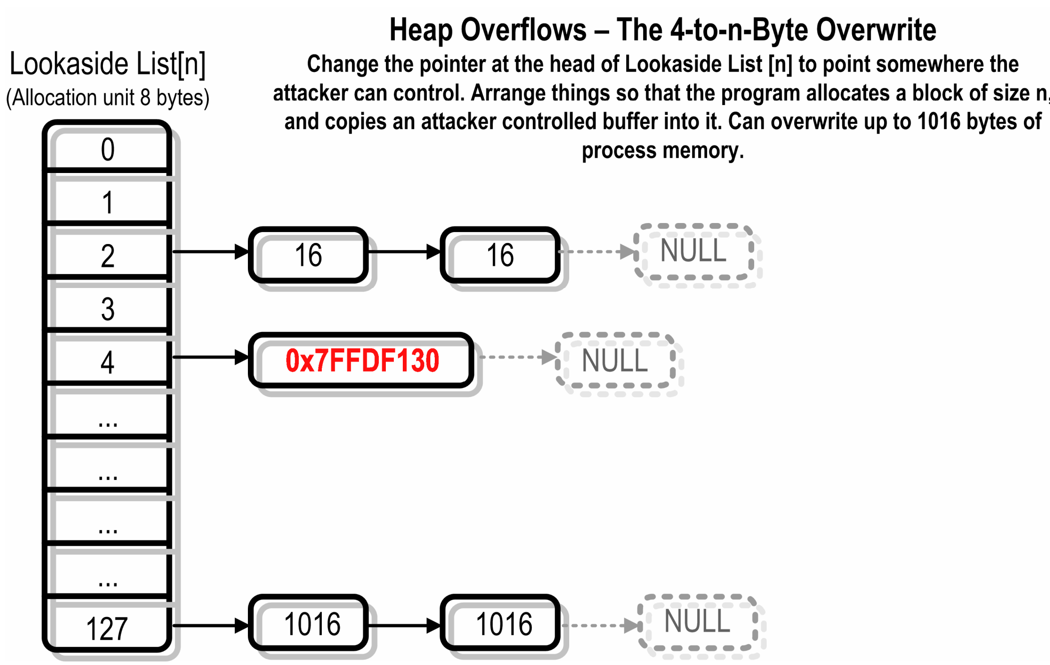
Another problem with the new checks is that they do not cater for a heap optimization feature called lookaside lists. Normally, new blocks are allocated from a freelist. During this allocation, the block is deleted from the freelist - at which point the safe unlinking check is performed and the heap cookie is checked for corruption.

Lookaside lists optimize re-allocation, by keeping track of small, recently freed blocks, allowing those blocks to be handed out quickly in response to new requests without traversing the freelist. Manipulating the lookaside list directly to provoke a "~1K overwrite", also known as the "4-n-byte overwrite" was already a public technique, and is documented in *Reliable Windows Heap Exploits*[40]. That attack first

[38] Above, note 35
[39] M. Conover, *XPSP2 Heap Exploitation.ppt*, available at http://www.cybertech.net/~sh0ksh0k/heap/
[40] In Conover / Horovitz, above, note 35

uses a 4-byte overwrite, as described above, to corrupt the lookaside list – but this attack would be prevented by the safe unlinking checks.

However, Anisimov[41] documents a newer technique which extends this to take advantage of an implementation problem with heap cookies – there is no integrity checking when blocks are allocated from a lookaside list[42]. In certain overflow scenarios the Flink pointer of a neighbouring block can be overwritten with an attacker-specified pointer. If that block is already in a lookaside list then on the second allocation request for a block of the selected size HeapAlloc() will return the pointer to up to 1016 bytes of attacker-chosen memory. The attacker then causes a copy operation that references this pointer and overwrites a large chunk of memory anywhere in the process address space. After the ~1K overwrite there are several possible exploitation techniques. Although this exploit approach is not guaranteed, Anisimov claims to have used it to create a working exploit for at least one real-world heap overflow.



As a final note, a new performance-focused heap technology called the Low-Fragmentation Heap could potentially complicate exploitation in many scenarios since it uses a 32-bit key in the chunk header instead of the current 8-bit cookie, but virtually no applications appear to be using it[43].

[41] A. Anisimov, *Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass*, 2005, online at http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf
[42] Also described by Conover, above, note 39
[43] Above, note 39

SEH Security

Exception handlers have traditionally been one of the most abused features in Windows exploitation. From a hacker's point of view, the exception handler is code that will be run *after* something dangerous has happened, which makes it fragile. One of the generic exploitation patterns is to overwrite a pointer to an exception handler with the location of malicious code, and then to provoke (or just wait for) an exception.

The definitive introduction to structured exception handling was written by Matt Pietrek back in 1997[44], and is still highly recommended background reading.

Windows exception handlers are particularly prone to abuse, not least because the EXCEPTION_REGISTRATION structure used by Structured Exception Handling (SEH) for each function is stored on the stack, allowing it to be overwritten by stack-based overflows. Here is an example from Microsoft[45]:

```
int vulnerable5(char * pStr) {
    char buf[32];
    char * volatile pch = pStr;

    strcpy(buf, pStr);
    return *pch == '\0';
}

int main(int argc, char* argv[]) {
    __try { vulnerable5(argv[1]); }
    __except(2) { return 1; }
    return 0;
}
```

As can be seen, main() defines a local __try / __except handler. This will caused an EXCEPTION_REGISTRATION structure to be created in the stack frame of the main() function. When main() calls vulnerable5(), the stack frame for vulnerable5() will be below the main() frame, meaning that buf[] can overflow *past* its own saved return address and overwrite main()'s exception handler. Now, because an exception is raised *before* vulnerable5() returns, the stack cookie is never checked and the compromised exception handler is run.

Another problem is that the Unhandled Exception Filter (UEF) (the exception handler of last resort) is stored in a fixed location for each Windows version[46]. Attacking the UEF is a well-known tactic when only a small overwrite can be achieved, for example with heap unlinking (such as Halvar's 4-byte overwrite, discussed in *Reliable Windows Heap Exploits*[47]).

In Windows XP, some improvements were introduced. The first was zeroing the CPU registers before an exception handler is called. This is in response to "trampoline attacks" (discussed also above) where

[44] M. Pietrek, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, in Microsoft Systems Journal, January 1997, online at http://www.microsoft.com/msj/0197/exception/exception.aspx
[45] B. Bray, *Compiler Security Checks In Depth*, 2002, available at http://msdn.microsoft.com/
[46] More accurately, each KERNEL32.DLL version
[47] Above, note 35

attackers can analyse the contents of registers at the time of the fault, and then use an exception handler to “bounce” off a known address which contains a relative call, such as “CALL EAX”.

The second major improvement was to deny the execution of exception handling code on the stack. This was to prevent a well-known technique which involved overwriting a stack-based SEH structure to redirect execution back into shellcode in the overflow data. This approach was used by the Sasser worm for Windows 2000 targets[48].

Microsoft’s proposed addition to the above protection, introduced in Visual Studio .NET 2003, was Safe Exceptions, or the /SAFESEH switch. In summary, Safe SEH works by producing a list of *all* valid exception handlers at the time the application is compiled and linked. When an exception occurs at runtime, the exception handler being used is only run if it is pre-registered – in other words trying to “re-wire” exception handlers during a control-flow attack shouldn’t work.

Microsoft also introduced a new internal function to NTDLL.DLL called RtlIsValidHandler to attempt to detect when control was about to be dispatched to attacker code.

Unfortunately, some unusual implementation problems in Windows 2003 SP0 greatly reduced the initial effectiveness of this feature. David Litchfield outlined several possible attacks in 2003[49], and pointed out a key implementation error. Unregistered exception handlers would still be run as long as they were *outside* the address range of a loaded module, to allow for run-time code generation. While the specific check remained in place to prevent the system from running exception handlers from the stack, there were no such protections applied to the heap, or to any other area of memory.

For example, Litchfield described an attack where a few bytes of static data in a memory mapped file (FF 55 30) are interpreted as an instruction when execution is directed to that address (it forms the processor opcode for CALL DWORD PTR[EBP+0x30]), and will redirect execution back to the EXCEPTION_REGISTRATION structure that is under attacker control. This kind of “dual use” approach to data / opcodes has led to several interesting exploits over the years.

Litchfield also described an attack where an existing exception handler could be abused as a trampoline to redirect control flow after the EXCEPTION_REGISTRATION structure overwrite. The handler referred to in that article was actually __except_handler3, which is the key function involved in the Visual C++ implementation of SEH. The Visual C++ runtime library layers additional functionality on top of the “raw” exception handling machinery provided by the OS, which is all well described in Pietrek’s article[50].

In Windows XP, SP2 and Windows 2003, SP1, Microsoft significantly improved the checking that is performed during the exception dispatch process. Unless software DEP has been deactivated, it is no longer possible to dispatch to an exception handler that is on a non-executable memory page, obviating the first attack described above using “raw” exception handling. Additionally, extra validation checks have been added to __except_handler3, which obviate the second attack using the C++ runtime library. It is important to note that the improved RtlIsValidHandler and the brand new __ValidateEH3RN provide protection against almost all types of SEH abuse even without CPU NX support and irrespective of whether or not the application has been compiled with /SAFESEH[51].

---

48 P. Ferrie, F. Perriot, *Mostly Harmless*, in Virus Bulletin, August 2004, online at http://pferrie.tripod.com/vb/sasser.pdf
49 Above, note 30
50 Above, note 44
51 Based on the author’s reverse analysis of ntdll.dll on Windows XPSP1, SP2, Windows 2003 SP0, SP1, publication of full findings pending.

.

PEB Randomization

The Process Environment Block is a structure which contains various pieces of useful information about the currently running process which code might need to access at runtime. The PEB is not well documented, but some more information can be found, for example, in an article from Relsoft Technologies[52].

One very small change in Windows XP Service Pack 2 was the slight randomization of the base address for the Process Environment Block (PEB), which has traditionally been 0x7ffdf000 for all NT based systems. The randomized locations vary only by a few pages, but that is enough to confuse shellcode that uses hardcoded addresses. For example, here is the start of the PEB from a random process on an XPSP2 machine:

```
Command - Pid 4020 - WinDbg:6.4.0004.3
0:006> !peb
PEB at 7ffd5000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
    Ldr                       00241e90
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00241f28 . 002430f0
    Ldr.InLoadOrderModuleList:           00241ec0 . 002430e0
    Ldr.InMemoryOrderModuleList:         00241ec8 . 002430e8
            Base TimeStamp                     Module
          400000 42796705 May 05 02:21:25 2005 C:\Program Files\iPod\bin\iPodService
        7c900000 411096b4 Aug 04 09:56:36 2004 C:\WINDOWS\system32\ntdll.dll
        7c800000 411096b4 Aug 04 09:56:36 2004 C:\WINDOWS\system32\kernel32.dll
        77e70000 411096ae Aug 04 09:56:30 2004 C:\WINDOWS\system32\RPCRT4.dll
        77dd0000 411096a7 Aug 04 09:56:23 2004 C:\WINDOWS\system32\ADVAPI32.dll
        74ae0000 4110969d Aug 04 09:56:13 2004 C:\WINDOWS\system32\CFGMGR32.dll
        77920000 411096b0 Aug 04 09:56:32 2004 C:\WINDOWS\system32\setupapi.dll
        77c10000 41109752 Aug 04 09:59:14 2004 C:\WINDOWS\system32\msvcrt.dll
        77f10000 41109697 Aug 04 09:56:07 2004 C:\WINDOWS\system32\GDI32.dll
        77d40000 42260159 Mar 02 19:09:29 2005 C:\WINDOWS\system32\USER32.dll
        774e0000 41e78916 Jan 14 09:55:50 2005 C:\WINDOWS\system32\ole32.dll
        77120000 411096f3 Aug 04 09:57:39 2004 C:\WINDOWS\system32\OLEAUT32.dll
        5b860000 411096ac Aug 04 09:56:28 2004 C:\WINDOWS\system32\NETAPI32.dll
        77c00000 411096b7 Aug 04 09:56:39 2004 C:\WINDOWS\system32\VERSION.dll
        76b20000 411096c7 Aug 04 09:56:55 2004 C:\WINDOWS\system32\ATL.DLL
        10000000 427966ed May 05 02:21:01 2005 C:\Program Files\iPod\bin\iPodService
          640000 427966eb May 05 02:20:59 2005 C:\Program Files\iPod\bin\iPodService
        20000000 411096b9 Aug 04 09:56:41 2004 C:\WINDOWS\system32\xpsp2res.dll
        76c30000 411096b9 Aug 04 09:56:41 2004 C:\WINDOWS\system32\WINTRUST.dll
        77a80000 41109691 Aug 04 09:56:01 2004 C:\WINDOWS\system32\CRYPT32.dll
        77b20000 411096e3 Aug 04 09:57:23 2004 C:\WINDOWS\system32\MSASN1.dll
0:006>
```

The PEB is an interesting target for hackers for a few reasons.

Locating the Relative Virtual Address of API functions

The PEB contains pointers to several important structures – for example a list of loaded modules and their base addresses. Shellcode usually needs to access functions from the Windows API; so the memory

[52] A. Ionescu, *Introduction to NT Internals, Part 1*, 2004, http://www.relsoft.net/Articles/Process/part1.pdf

addresses of those functions must be discovered before they can be called. Simplified, one traditional method of doing this is as follows:

1. Access the loader data (Ldr) in the PEB at offset 0x0c
2. Walk one of the module lists in the loader data to find the module's base address
3. Read the module's export table to find the RVA of the desired function

The simplest way to proceed from there is to find addresses for GetProcAddress() and/or LoadLibrary(), which can then be used to supply the addresses of all other required functions.

More detailed descriptions of walking the LDR can be found online[53] and code to locate LoadLibrary by accessing the PEB and then locating KERNEL32.DLL is commonly available[54]. By rebasing the PEB, the initial offset is no longer constant. However, this should not be taken to imply that randomising the PEB is a solution to function address location. Other methods do exist which are not reliant on the PEB, for example sequential scanning for the module's prolog and parsing the module's export table directly. Or, more simply, the PEB can always be accessed by reference to the FS selector, ie:

```
MOV EAX, FS:[30h]
```

Overwriting Critical Pointers

Some newer exploits are specifically targeting interesting pointers within the PEB itself – for example the recent public release of an exploit for an ASN.1 bitstring vulnerability[55] uses a hardcoded address of 0x7ffdf020 for the pointer to FastPEBLockRoutine(). The FastPEBLockRoutine is called when code wants to lock the PEB so that it can be modified – and it happens to be called during exception handling. In other words this exploitation trick is another variation of abusing code that is run after memory corruption has occurred.

Interestingly, Microsoft have removed some of the interesting pointers from the PEB under Windows 2003, so many exploits that use these methods are not effective for Windows 2003 systems. Specifically, the pointers FastPEBLockRoutine() and FastPEBUnlockRoutine()at PEB offset 0x020 and 0x024 have been removed.

Running Shellcode in the PEB

One last reason the PEB is interesting to attackers is because the memory pages where the PEB resides are both Writeable and eXecutable. Since the PEB has always been in a fixed location, attackers have the option of copying their shellcode to a known address in the PEB and then transferring execution to that location. This technique can be useful when an arbitrary memory overwrite is possible but the payload is difficult to locate – such as for heap overflows. For this reason, copying shellcode into the PEB and then using an overwritten pointer which is called during exit is the currently preferred method for universal and reliable heap exploitation on Windows 2000 and Windows XPSP1[56]. PEB randomization can complicate this technique slightly, but since the address change is only very slight, an attacker can still "guess" a safe location to copy the payload with high reliability.

[53] E.g. A. McDonald, *FAR (Function Address Retrieving)*, in blog arnold.mcdonald, http://arnold.mcdonald.free.fr/php/Main.php?p=1007, 2004, accessed June 20, 2005
[54] In McDonald, above, note 53, also in LSD, *Win32 Assembly Components*, 2002, http://www.lsd-pl.net/documents/winasm-1.0.1.pdf
[55] S. Eclipse, *kill-bill*, 2005, http://www.phreedom.org/solar/exploits/msasn1-bitstring/
[56] In Conover / Horovitz, above, note 35

Having earlier examined PaX's ASLR, the limited changes to the PEB base seem somewhat tame; additionally, because of the limited scope of the randomization there is a reasonable chance that the PEB will be based at 0x7ffdf000 in any case. However, for a "low cost" change the PEB randomization feature measurably complicates exploitation, making some well-known techniques less reliable.

Pointer Security

As mentioned briefly above, Microsoft have removed some pointers from the PEB in Windows 2003, making attempts to leverage small overwrites (such as the heap 4-byte overwrite) more difficult. This approach has also been extended to the UEF pointer in Windows XP SP2 and Windows 2003 SP1, and the VEH (Vectored Exception Handler) pointers in Windows 2003 SP1 only.

An additional level of security has been added to these pointers by encoding them in memory (an XOR with a modified random seed). Legitimate dispatches to the UEF or a VEH will first call RtlDecodePointer to obtain the real pointer value. An attacker, however, is unable to predict the random seed – whatever value the attacker chooses to overwrite the pointer with, it will almost certainly be decoded into garbage, and the process will crash rather than jump to the attackers shellcode. The pointer encoding algorithm is described in depth in a paper by members of the Xfocus group[57]. The paper is in Chinese, but translates reasonably well with online translation services.

The new pointer security is a tactical measure, but currently it works well to complicate the basic, well known attack patterns designed to leverage small overwrites into fully fledged code execution.

NX Memory and Hardware Enforced DEP

Although the basic operation of NX under Windows is fairly intuitive, it is worthwhile quickly summarizing the major concepts involved.

Windows Data Execution Protection (DEP) is an architectural concept, implemented in software, which surrounds the way in which Windows will "perform additional checks on memory to help protect against malicious code exploits". Hardware enforced DEP is the combination of DEP and CPU NX support. Software enforced DEP consists of the changes to stack and heap behaviour discussed above.

Regarding NX memory pages, one common source of confusion arises from the fact that Windows *already* supports memory page permissions via the VirtualProtect() API call, allowing pages to be marked as non-executable - for example PAGE_READWRITE. These settings, however, only apply to the Windows virtual memory manager – in other words they are not reflected in the physical page table entries (PTEs) and are thus invisible to IA-32 CPUs.

CPU support for marking a page as non-executable is a pure hardware feature, which can potentially be used by any operating system that supports it. First to market for the Windows desktop, AMD x86_64 processors support an "NX bit", with Intel following suit with "XD" (for eXecute Disable) on i915 based Pentium 4 processors. For the UNIX world, Alpha, PowerPC and Sparc have supported it for some time.

---

[57] funnywei, jerry, *Windows Xp Sp2* [illegible], available at http://www.xfocus.net/articles/200412/762.html

Deeper details of how NX is actually implemented are definitely beyond the scope of this paper, interested readers should consult the programmer's documentation for their CPU.

It is important to know, though, that 32-bit operating systems can only access the Intel NX functionality when running with Physical Address Extension (PAE) mode turned on. PAE is an address extension technology first introduced in the Pentium Pro processor, which uses 36-bit physical addressing to allow high performance systems to use more than 4GB of RAM[58]. However, PAE is not generally turned on for desktop Windows OSes, and some applications and drivers have compatibility issues when running under it.

Finally, the software side of the equation is covered in Microsoft documentation[59]. DEP can be enabled (or disabled) globally, and can be selectively disabled per-process. When an attempt to execute an instruction on an NX page is detected, an exception (STATUS_ACCESS_VIOLATION (0xc0000005), type 8) will be raised. In normal circumstances that exception would be unhandled, and the process will then terminate itself.

Raising a standard exception is perhaps the most controversial decision in the Windows DEP implementation. When exception handling is invoked, a significant amount of user-land code is run before the process actually terminates itself, and that code has already been intensely scrutinized by attackers. As some researchers have pointed out[60], it would be more secure to trap this exception in the kernel, before it is even transferred to the KiUserExceptionDispatcher() function in user mode. However, it is clear that such an implementation would violate the general principles of Windows structured exception handling.

Previously discussed methods for bypassing Windows stack and heap protection are often generically applicable to a full NX implementation – in particular classic return-into-library methods which attack control-flow by using injected data with existing code (such as fake stack based function arguments). For this reason, NX in general and Windows Hardware Enforced DEP should not be seen as a panacea.

However, the combination of stack / heap protection *and* NX complicates the standard exploitation patterns significantly. Stack protection will apply to all Windows applications and system components in XPSP2 and W2K3, but it only applies to applications which are compiled with the /GS option - so most third party applications will still be vulnerable. Safe SEH is rarely used by third-party application developers, although Windows components use it on XPSP2 and W2K3. Heap protection is implemented as changes to the heap allocation and deallocation routines, so it will apply to all applications running on XPSP2 and W2K3.

---

[58] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, 2005, section 3.8, available at http://developer.intel.com/design/pentium4/manuals/index_new.htm
[59] S. Andersen, V. Abella, *Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3*, 2004, http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx
[60] In Litchfield, above, note 30

Summary

Overall, the security changes in XPSP2 and W2K3 are much further reaching than the few discussed here, and can mitigate attacks at many levels before the anti-exploitation features come into play. The anti-exploitation technology itself represents a giant improvement over earlier Win32 operating systems. Yet, even considering the combination of all the new Windows technologies, some attacks remain viable. These attacks are mainly due to the implementation problems previously discussed, and would be greatly constrained by wider adoption of hardware NX and improved integrity checking in the heap allocation and free routines. For now, the combination of stack cookies and the new SEH security has not been publicly bypassed, although heap overflows remain dangerous, with documented attack patterns in the public domain.

The table below summarises example application-generic attack patterns that are still effective against the new anti-exploitation technologies in W2K3 and XPSP2. In the table, we have ignored the fact that PEB randomization is not present on W2K3, since it makes little difference; although shellcode developers will need to avoid module location code that uses hardcoded PEB offsets.

Target OS	Stack Overflow	Heap Overflow
XPSP2 / W2K3 • /GS compiled binaries • Safe Unlinking • (PEB Randomization) • Safe SEH	No generic attack patterns currently known.	Safe Unlinking bypass or Lookaside List overwrite + 1k overwrite → multiple techniques to keep control Author: Conover[61] Ease: Very Hard Reliability: Low Re-usability: Medium
XPSP2 / W2K3 • /GS compiled binaries • Safe Unlinking • (PEB Randomization) • Safe SEH • NX (Hardware DEP)	No generic attack patterns currently known.	Safe Unlinking bypass or Lookaside List overwrite + 1k overwrite onto stack → ret-libc Author: Anisimov[62] Ease: Very Hard Reliability: Low Re-usability: Low
Ease:	Extremely unscientific assessment of the difficulty of constructing the attack, also includes some consideration of the likelihood that a totally arbitrary vulnerability will be exploitable with this pattern	
Reliability:	Similarly unscientific assessment of the reliability of a correctly written exploit against a vulnerable target	
Re-usability	Likelihood that the attack could be re-used to exploit another vulnerable application with minimal modification	

[61] Above, note 35
[62] Above, note 41

3rd Party Anti-Exploitation Technology for Windows

For many Windows users, upgrading to XPSP2 or W2K3 is impossible for application compatibility or commercial reasons. This means that those users are unable to benefit from the Windows anti-exploitation features discussed above. To exacerbate the issue, operating systems such as Windows NT 4.0 – which is still widely deployed on production systems in many industry verticals – have passed beyond the limits of Microsoft's support lifecycle. This means that no patches will be provided for new security issues unless the user purchases an expensive extended support contract; and even then there is no guarantee that a patch will be produced at all. For these users the only options are to turn to 3rd party solutions, or live with existence of critical vulnerabilities.

To meet the need for backwards compatibility, as well as to address some of the perceived shortcomings in Microsoft's own protection mechanisms, 3rd party vendors have devised several approaches to generic anti-exploitation.

To avoid any possible controversy, only the major techniques used by 3rd party applications will be covered – no specific products will be mentioned. Other researchers have investigated individual products, for example an excellent article by Jamie Butler and two anonymous authors in Phrack 62, 0x05[63] and a Blackhat presentation by iDefense[64]. It must be noted that an unfortunate limitation of this non-specific approach is that vendor-specific implementation flaws cannot be examined. For this reason, we recommend at least the above links for further reading on some of the implementation flaws that have been identified in various products.

The first thing to note about 3rd party solutions is that almost all current techniques are exploit focused (see Fig 1). Vulnerability focused solutions tend to require either compile-time modifications or changes to core operating system functionality. Because the major operating system components such as the virtual memory manager the core heap routines like RtlAllocateHeap() are owned by Microsoft, any changes to their behaviour would be both difficult and fragile. Compiled-in solutions would require changes to the compilers used to build the core Microsoft system components, which is obviously impractical.

[63] J. Butler, Anon, Anon, *Bypassing 3rd Party Windows Buffer Overflow Protection*, Phrack 62, Article 5, 2004, available at http://www.phrack.org/show.php?p=62&a=5
[64] P. Silberman, R. Johnson, *A Comparison of Buffer Overflow Prevention Implementations and Weaknesses*, 2004, available at http://blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf

Userland API Hooking

This is the most common implementation of generic buffer overflow detection in commercial products.

First, let us recap what is meant by function hooking. At the time when the protection mechanism is loaded, key API functions are modified, and a new prolog and epilog are "wrapped" around the original code.

When used for buffer overflow detection in its simplest form, this new code runs at the entry point of the function and performs a check something like:

1. What is the memory address of my caller?
2. If that address is in writeable memory, assume buffer overflow

Following the paradigm that code should never be executed in writeable memory.

For performance reasons, only a selection of userland APIs are hooked, based on the observation that the vast majority of real-world shellcode will contain calls to one of these key APIs – for example the LoadLibrary family and GetProcAddress family, which are used to obtain the addresses for many other functions. To use a simplistic analogy, these checks are like booby-traps. As soon as shellcode calls a "trapped" API, the protection software can terminate the process – without calling long and vulnerable Windows exception handling routines. It must be noted, of course, that shellcode that doesn't call any trapped functions can run unimpeded.

The advantage of this approach is that it is backwards compatible across all NT based systems, and can protect the heap as well as the stack. Another key advantage is that it bypasses structured exception handling – as should be clear from the previous section, this mitigates many possible attacks.

There are a few problems however, covered well in Butler's phrack article[65], which mean that an attacker who has analysed the protection method will almost always be able to bypass them under lab conditions. However, even fairly simplistic API hooking systems can provide very effective protection against the overwhelming majority of exploits currently seen in the real world.

[65] Above, note 63

Return address checks

Several checks can be made to ensure that the return address presented to a hooked API is legitimate. One refinement which is added by various products is to ensure that the function's return address immediately follows a call or jump, known as "ret follows call".

```
push    eax                  ;this opcode is not a control transfer
lea     eax, [ebp+var_4]     ;so this is not a valid return address!
push    eax                  ;
push    0FFFFFFFFh           ;
call    HookedAPI            ;call to a hooked function
mov     ebx, eax             ;valid return address (ret-follows-call)
```

During an attack such as a return-into-libc, the attacker aims to execute a library function with their own malicious function parameters. However, at the point where the API is about to return the attacker normally wants to keep control of execution. To do this, they will supply a fake return address as part of the malicious stack frame, and execution will be transferred to that address after the API function has completed. It is very likely that the attacker wants to transfer execution to some random address that contains more malicious code (or at least attacker chosen code) and that this code is outside the normal control-flow of the program.

To complicate such attacks, the protection software checks the instruction *immediately preceding* the proposed return address to ensure that it is a control transfer instruction – if it is not then there is no legitimate way execution could have reached the hooked API. By policing this fairly simple rule, protection systems are able to further complicate exploitation.

Another obvious check is to ensure that the return address lies within non-writeable memory. As discussed above, StackShield implemented a basic version of this check called "return range" checking, which was quick (since it used easily calculated ranges) but vulnerable to abuse[66]. Some enhancements to this approach are discussed below.

Forward Emulation

An interesting refinement to the API hooking technique is to perform limited x86 emulation, to attempt to detect multi-instruction trampoline sequences. Much exploitation research has been done into ways to keep control of execution while "bouncing" off multiple instruction sequences. Early "trampolines" aimed to transfer execution to known opcode sequences which would return execution to the payload such as CALL EAX or PUSH EAX / RET. Although these attacks were initially devised to transfer execution to shellcode when the exact address of the payload was unknown, they also have the effect of confusing 3rd party protection. For example, API hooks that attempt return range checking to verify that return address are outside the stack or heap will be fooled by trampoline sequences in a module that bounce execution directly back onto the stack. Initially, simple checks could be used to detect these kinds of trampolines, mostly looking for opcode signatures, but those checks are, in turn, defeated by increasing the complexity of the trampoline code being used.

[66] For example, in Richarte, above, note 6

As an attack feasibility study, eEye researchers presented an automated tool (EEREAP)[67] at BlackHat USA 2004 which was designed to find instruction chains that would return execution to a given point. EEREAP was able to find chains that would return execution to an attacker buffer which were up to 14 or 15 instructions, or even longer – obviously this would frustrate simplistic "trampoline" analysis.

X86 emulators as EEREAP can themselves be used by protective mechanisms to follow execution chains at the proposed return address to determine if execution *could* land back in writeable memory, anticipating that attackers will soon move away from simple byte sequences like CALL reg, or PUSH reg / RET. Although forward emulation systems significantly complicate exploitation, there is a performance cost which must be borne in mind.

Windows PAGEEXEC

At least one product has attempted to use PaX's PAGEEXEC IA-32 technique to provide emulated NX memory, using the CPU Translation Lookaside Buffer in much the same way as PaX, with some optimizations. This technique is discussed briefly above and in much more depth in the PaX documentation[68]. Unfortunately, this emulated NX technique is quite slow, so this is not a common approach, and the only commercial implementation we examined had severe limitations.

Windows ASLR

As mentioned above, one or two 3rd party products have attempted to implement PaX ASLR style protection for Windows. Unfortunately, although Windows PE-COFF binaries support relocation, many Windows subsystem and kernel components do not. In particular, KERNEL32.DLL, USER32.DLL and NTDLL.DLL cannot be re-based without jumping through hoops to trick the loader. This makes return-into-libc style protection much less effective, since most "interesting" functions lie within those modules, as well as many useful opcode chains that can be used as trampolines. Additionally, there are extremely troublesome compatibility issues with layout randomization, so overall it cannot currently be recommended for Windows.

Kernel API hooking

Most 3rd party solutions also hook a selection of kernel APIs.

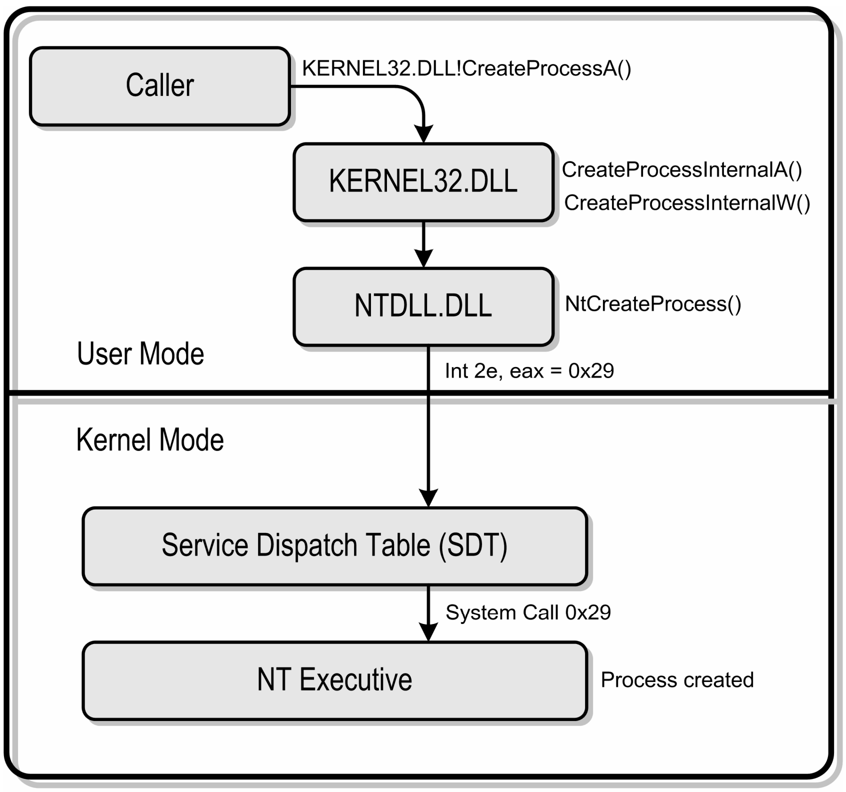
Before we discuss the hooking approaches themselves, a quick note on implementation. Rather than "hooking" kernel functions, a more appropriate term would probably be "redirecting", since the functions themselves do not need to be modified. Instead, a kernel structure called the Service Dispatch Table (SDT) can be re-wired. The SDT is simply a lookup table which translates "syscalls" to kernel function pointers. By modifying the SDT pointers the inspection code can be run instead of the "real" function. Assuming no attack is found, control is passed directly to the destination function. This approach could also be used in userland hooking, by modifying the Import Address Table – but it would be next to useless, since an attacker would simply jump directly to the function itself. Transferring control to the

[67] D. Soeder, R. Permeh, Y. Ukai, *Advanced Return Address Discovery using Context-Aware Machine Code Emulation*, presented at Blackhat 2004, available at http://blackhat.com/presentations/bh-usa-04/bh-us-04-soeder/bh-us-04-soeder-up.ppt
[68] Above, note 19

kernel, however, is always achieved through interrupts or call gates, so jumping directly to kernel code is not possible.

There are two key reasons to hook functions in the kernel. The first is ubiquity – hooking a kernel syscall will allow *all* API calls that lead to that system call to be inspected, without the need to hook one of the many userland functions which might have the same effect – including variants such as ApiCallA() / ApiCallExA() for ANSI, ApiCallW() / ApiCallExW() for Unicode, or evasions such as directly calling one of the undocumented ntdll.dll versions of a kernel32.dll API like NtApiCall() or ZwApiCall().

The big downside to this method is that once operating in the kernel it becomes much harder to verify how the function was called. In other words, attempts to enforce the "caller in read-only memory" rule become very difficult, since by the time a kernel function is called, execution has normally passed through two or more "wrapper" functions, and the address of the original caller is several frames up on the userland stack. This is illustrated in the diagram below:



Windows API Call Abstraction (Windows 2000)

Some systems attempt to track the caller by "walking the stack". Recall that a stack frame usually (but not always) contains a saved frame pointer in EBP, so by following EBP we can find the stack frame of the function that called us. And by following the saved return address from *that* frame, we can find the previous caller, and so on.

This approach is trivial to defeat, however. If an attacker has control of the stack, they can create "fake" frames to defeat the checking mechanism. Overall, attempting to hook kernel system calls to enforce "caller in writeable memory" checks appears to be impractical.

Mitigation of malicious activity

The second reason to hook in the kernel, however, is to apply generic mitigation. Although slightly outside the scope of this paper as defined in Fig 1, mitigation techniques are an important part of 3rd party anti-exploitation systems.

Kernel-based mitigation approaches aim to police exactly which system actions can be taken by *any* process on the system – APIs called, registry keys accessed, files read or written and so on. The advantage to this approach is that it is able to enforce behaviour in all cases – both when there has been an attack which has successfully bypassed all anti-exploitation measures as well as for situations in which there has been no vulnerability exploitation at all. A good example of the second category is a Trojan horse, where the user has deliberately launched a malicious application. 3rd party vendors take two approaches to this enforcement, "known good" and "known bad" profiling.

"Known Good" profiling, which can also be loosely referred to as "sandboxing", attempts to define a full set of system calls that are accessible to a given application, and even to enforce acceptable parameters to those system calls – eg NtCreateKey might only allow an application to create keys in HKLM\Software\ApplicationA and not in HKLM\[...]\Run. Creating this set of permitted activities is known as "baselining" or "learning mode". In the real world, usability compromises are always taken, creating a "looser" baseline, which again opens the door for malicious activity, especially given that modern attackers are fully capable of reverse engineering these protection mechanisms.

"Known Bad" profiling attempts to select the most dangerous system calls such as WriteProcessMemory(), SetWindowsHook(), TerminateProcess() and then police all access to such calls. For example, almost no legitimate applications need to directly write into the memory space of another process, but it is a well known trick used by malware for a variety of purposes. Under Windows, many different malware techniques eventually result in a call to WriteProcessMemory(), so by controlling access to one API many attacks can be prevented.

Known good profiling offers better potential security in theory. In practice, however, obtaining a tight baseline of "good" actions which still allows the system to function correctly in all legitimate cases is extremely challenging, and creates a significant usability challenge.

Known bad enforcement can be applied much more rigorously, but does not cover all possible system calls. The advantage to known-bad systems is that they can be applied with minimal baselining and with negligible rates of false positives. In real world application, known bad enforcement aims to provide "good enough" protection, especially when used in combination with other technologies such as generic anti-exploitation, application-aware firewalling and heuristic traffic inspection.

When considering 3rd party systems which will be widely deployed across general purpose workstations, known-bad enforcement strikes a better balance between security and usability. For task-based systems like servers which change very rarely and have an extremely limited range of functions, known-good profiling may provide more rigorous security, if the specific implementation is suitable.

In any case, both approaches add a valuable layer of enforcement capability to the existing Windows security features.

Self Defense

The last point to note is that, since attackers may be aware of the operation of the third party software, it is quite possible that they will attempt to bypass its protection entirely. For example, software that uses predictable hooking preambles can easily be detected. Once the shellcode is "aware" that such protection exists it can attempt to bypass the mechanism. Butler's Phrack 62 article[69] analysed some early userland API hooking solutions which have few or no built-in defenses against reverse engineering or hook detection and evasion. This is not the case with all systems.

By using techniques borrowed from malware and anti-disassembly tools, defensive software is able to protect itself from such attacks. Polymorphic hooks make it impossible for malware to use "hook signatures" to detect the protective mechanisms. Guard pages can act like landmines to prevent sequential scanning – since an attempt to read a guard page will trigger a STATUS_GUARD_PAGE exception and the process can be terminated.

Defense against a local attacker is much more complicated. For malware that is running locally with the permissions of the logged-in user (such as most viruses and Trojan horses) there are a range of other possibilities. Full coverage of the current state of the art in attack and defense of 3rd party protection from a local attacker would require a paper of its own. However, some good introductory reading covering some newer attacks is available in Phrack 62[70] and an article by 3APA3A and offtopic[71]. These papers cover runtime process infection, "shatter" attacks[72] and scripting attacks that mimic user input (to disable the firewall, for example). Each of these specific attacks are mitigated in a handful of state-of-the-art solutions, but such generational advantages never last for long.

While it is not something that most commercial vendors are keen to admit, the simple fact is that it should always be assumed that there is a way to bypass any kind of protection once code is running locally. 3rd party solutions can install significant barriers in terms of self-defense, but due to the "defender's dilemma" an attacker will always be at an advantage. However, one simple step that can be taken to greatly complicate things for the attacker is to remove Administrator rights from users' day-to-day accounts.

Future Approaches

As can be seen from the discussion above, generic anti-exploitation technology is a complex field. Literally dozens of solutions exist; each designed to address certain kinds of exploitation. In many cases, these solutions have been bypassed, improved and bypassed again. It is to be expected that both the built-in Windows approach and 3rd party solutions will continue to improve on an evolutionary basis.

In comparison to the plethora of solutions that have been mooted and implemented to provide tactical coverage, relatively little revolutionary research has been done into addressing the problem of control-flow attacks as a whole, and none of these approaches have progressed past the prototype stage into general availability for either commercial or open source systems.

---

[69] Above note 63
[70] Rattle, *Using Process Infection to Bypass Windows Software Firewalls*, Phrack 62, Article 13, available at http://www.phrack.org/show.php?p=62&a=13
71 3APA3A, offtopic, *Bypassing client application protection techniques*, 2004, online at http://www.security.nnov.ru/advisories/bypassing.asp?l=EN
[72] B. Moore, *Shattering By Example*, 2003, online at http://www.security-assessment.com/Whitepapers/Shattering_By_Example-V1_03102003.pdf

Nevertheless, there are still a few research projects that aim to *generically* eliminate control-flow attacks. These approaches focus on the basic observation that all control-flow attacks work by adding new nodes to a control-flow graph, or executing existing code out of order by adding new edges. By enforcing control transfers such as entry and exit points, *all* control-flow attacks can be mitigated – at least in theory. Of course even if the design is sound, the implementations may not be. One thing that should be very clear from the discussion above is that many well designed solutions have been successfully and generically bypassed due to implementation flaws.

First, we examine two systems which attempt “sloppy” enforcement of correct control flow.

As mentioned above, PointGuard was an ambitious project which approached the problem by encrypting pointers while they are stored in memory. This approach aims to ensure the integrity of program control flow without attempting to understand or model it. Pointers that were modified at runtime by a control-flow attack would be converted into garbage values when decrypted, almost certainly provoking an access violation or crash. Unfortunately, PointGuard as described was a compiled-in solution, making it unsuitable for Windows (unless Microsoft choose to implement it themselves). Other papers identify certain limitations in the PointGuard implementation[73], and also propose a hardware based implementation.

A different approach that pre-dates PointGuard is Program Shepherding[74] which used a machine-code interpreter to provide runtime checking. Program Shepherding allows arbitrary restrictions to be placed on code execution, allowing for both tight and loose control flow policies.

By using “Restricted Code Origins”, “Restricted Control Transfers” and “Un-Circumventable Sandboxing”, security policies can be defined and applied to code at runtime. This flexible policy approach could allow end-users to choose their own balance between security and usability, which is an important factor for most organisations. Another advantage is that Program Shepherding should be more or less operating system independent, offering the possibility of a truly cross platform solution. It must be noted, however, that Program Shepherding still does not aim to understand the CFG for each given application, it aims instead to apply arbitrary security policies to the observed control flow at runtime.

On the negative side, a significant part of the potential protection offered by Program Shepherding can be replicated by lower-impact solutions. 3rd party anti-exploitation solutions can already implement the most valuable checks that were suggested by the Program Shepherding team, such as “ret follows call”, return range checking and enforcing module entry points. These checks can be performed at runtime with function hooking and code injection. The “Restricted Code Origins” checks are covered to a great extent (although not completely) by any approach that provides non-executable data segments and read-only code segments.

The biggest disadvantages of the Program Shepherding approach were its complexity and performance penalties. Since it is relying on a machine code interpreter to apply the security policy to code operations, the implementation would need to prove that the interpreter itself is immune to attack. Further, the performance penalties of Program Shepherding are relatively steep. Nevertheless, a system which is able

[73] N. Tuck, B Calder, G. Varghese, *Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow*, in Proceedings of the 37th International Symposium on Microarchitecture, December, 2004, online at http://www-cse.ucsd.edu/~calder/papers/MICRO-04-CodePointerProtection.pdf
[74] V. Kiriansky, D. Breuning, S. Amarasinghe, Secure execution via program shepherding, in 11th USENIX Security Symposium, 2002, online at http://www.cag.lcs.mit.edu/commit/papers/02/RIO-security-usenix.pdf

to implement standard, flexible execution policies across multiple operating systems is extremely commercially attractive, assuming the problems could be overcome.

The final, and most recent, approach examined is a research project on Control-Flow Integrity (CFI) from Microsoft Research, published in February 2005. CFI is described both in informal language[75] and formally[76] in separate papers.

CFI is different to the other solutions presented so far in that it *does* attempt to understand the CFG of protected applications. In summary, CFI works in two stages. The first is to determine the acceptable control flow of a given program. This can be performed by runtime analysis (profiling), source code analysis or static binary analysis.

Once the control-flow has been analysed and a CFG produced, the binary is then permanently modified with code that verifies control flow changes at runtime – in essence self-verification checks are added to the machine code of the program itself. In the example implementation, control transfers are verified with matching pairs of id tags, so that instead of “call location” an id tag is added to form “call id, location”. In this way the id tag can be “checked” when control is received at the destination address[77]. By performing verification in this way the system aims to prevent attackers from inserting new edges in the CFG – recalling that without adding a new edge no control-flow attacks are possible.

The initial CFI research is promising for several reasons:

- It is backwards compatible, since it uses binary instrumentation that can be applied to older versions of Windows
- The performance penalty is manageable. As an added advantage, with future modifications to the IA-32 machine code instruction set the performance impact could be virtually eliminated
- Because it uses simple binary instrumentation rather than a complex external reference monitor, the security provided by CFI can be formally analysed, an important factor for some organisations

There are some limitations to the CFI approach – for example it is measurably weakened on systems that do not support some form of non-executable data. Another issue is that a static analysis approach would most likely present major compatibility issues for the growing incidence of packed and encrypted binaries from commercial vendors.

Nevertheless, CFI seems to offer the best current prospect for a final, “all-in-one” solution to control-flow attacks as a general problem. Unfortunately, the history of anti-exploitation technology leaves little justification for optimism in this regard, so hostile examination of a working implementation is crucial

In summary, PointGuard attempted a *de facto* enforcement of control-flow by protecting pointers to code destinations. Program Shepherding attempted to apply predefined control-flow policies in a trusted machine code interpreter without analyzing each specific binary. CFI aims to model the static control-flow graph of each program, and then produces an instrumented binary which will internally detect CFG diversions at runtime.

[75] M. Abadi, U. Erligsson et al, *Control-Flow Integrity*, 2005, available at http://research.microsoft.com/research/pubs/view.aspx?tr_id=868
[76] M. Abadi, U. Erligsson et al, *A Theory of Secure Control Flow*, 2005, available at http://research.microsoft.com/research/pubs/view.aspx?tr_id=867
[77] Actually, this is somewhat simplified. Because it is not possible to modify the processor CALL opcodes, the id tags are passed via CPU registers, in one of several possible opcode sequences.

Conclusions

Based on the research and analysis above, we submit the following conclusions:

1. Most Anti-exploitation technology provides less security than claimed
2. 3rd party solutions will remain attractive mid-term
3. Future solutions will unify anti-exploitation and mitigation approaches
4. Anti-exploitation is imperfect and should be a last resort

Most Anti-exploitation technology provides less security than claimed

In the examination of the new Microsoft technologies, the most striking fact was that each technology contained implementation problems that significantly weakened the offered protection. Some of these implementation choices were made for legacy reasons, to preserve uniformity of exception handling, for performance or for compatibility – but whatever the reasons, all of the Windows features could be implemented more securely.

However, this seems to be far more than a Microsoft problem; almost all of the open source solutions examined above provided less than their implied security targets. However, such shortcomings are only apparent on close technical examination. These problems were sometimes caused by implementation errors, but, mostly, particular choices were made for usability and compatibility. Finally, other works show that this is also true of many of the 3rd party Windows technologies, which we have chosen not to examine individually.

Another important point to consider is that in more than one case, the protection technology introduced critical *new* security vulnerabilities, thus reducing the overall security of systems rather than increasing it. This has been true of several 3rd party solutions for Windows, and was also one of the key reasons that the PaX project was abandoned.

This situation further complicates the potential widespread adoption of Windows anti-exploitation technology. Anti-exploitation technology is already difficult for most end-users to understand, let alone differentiate between the security profiles offered by various approaches. Technology that promises more than it delivers serves only to undermine users' trust.

3rd party solutions will remain attractive mid-term

Generic anti-exploitation technology for Windows is a relatively new field, especially when compared to the work done on various UNIX platforms. With Microsoft now entering the game, and committing to "Isolation and Resiliency" as one of their four pillars of security[78], the question must be asked – is there a future for 3rd party anti-exploitation software? Microsoft seem to have all the advantages – size, access to the source-code, access to the compiler and control of the native API.

Based on the discussion above, we would submit that the answer must still be yes – 3rd party anti-exploitation technology will continue to be valuable for the mid-term.

[78] B. Gates, *Microsoft Progress Report: Security*, 2004, available at http://www.microsoft.com/mscorp/execmail/2004/03-31security.asp

It is not proposed that 3rd party solutions replace Microsoft solutions – most particularly because there are certain areas (such as compiled-in systems like /GS and heap management routines) that are almost totally inaccessible to 3rd party developers. However, 3rd party systems have several advantages when considering systems that are exploit focused:

Backwards Compatibility

Because 3rd party systems are typically injected into process and kernel space, they can be easily designed to be backwards compatible, at least for the NT kernel operating systems (Windows NT 4.0, Windows 2000, Windows XP, Windows 2003). Systems that require binary recompilation or major changes to core system components do not have that advantage, and usually require a forklift upgrade.

Agility

Most 3rd party systems can operate on a much shorter “generation” cycle than core operating system changes by Microsoft. This is particularly important for anti-exploitation techniques, since the field has shown a constant history of attack and counter-attack between the creators of anti-exploitation solutions and attackers.

Directness

3rd party vendors can afford to be more direct in their approaches to protection – for instance in the event that a buffer overflow is detected a 3rd party system will typically terminate the process or thread immediately, without raising an exception and taking the risk that structured exception handling has been compromised.

Innovation

Because competition between 3rd party solutions is high – there are dozens of different commercial and open source solutions that offer generic anti-exploitation technology – there is a burning drive to create innovative technology. While this sometimes has the lamentable effect of creating “marketing wars” where the battle is lost or won in the way the products are promoted, it also encourages innovators to create a competitive advantage by building solutions that are simply, technically, better.

Future solutions will unify anti-exploitation and mitigation approaches

One of the key limitations to anti-exploitation technology is that it is only applicable to attacks that leverage a technical vulnerability by using a control-flow attack. Historically the most dangerous class of attacks, this attack pattern was most public from 2001 to 2004, when worms were causing frequent and widespread damage. However, in the first half of 2005, the focus has been shifting to a large class of attacks such as Spyware, Phishing and Trojan Horses which all eventually rely on computer misuse. These attacks are known, tongue-in-cheek, as "layer 8" attacks[79], since they rely on tricking a user.

Anti-exploitation technology is ineffective in the face of these layer 8 attacks. However, many anti-exploitation approaches can be extended to also provide mitigation of the malicious effects. API profiling, "sandboxing", virtualization, and outbound traffic control are all valuable mitigation techniques which could be effective against worms and targeted network attacks as well as layer 8 attacks.

Since many 3rd party anti-exploitation technologies contain a kernel component, they are well placed to enforce either selective denial of dangerous APIs or known-good API enforcement based on policy or pre-defined behaviour profiles. Note that, despite attempts by marketing departments to expropriate the term, solutions that take the latter approach fall short of true sandboxing - since they do not create a truly virtualized execution environment.

Another area in which mitigation technologies can be effective are for pure-data attacks such as those introduced by poor web application programming – notably SQL Injection and Cross-Site Scripting (XSS). These attacks do not exploit a control-flow vulnerability, but use malicious data as input to scripts. Technologies that can mitigate the effects of an attack rather than block the attack itself are effective here, since network based technologies to generically detect and prevent these kinds of attacks face a daunting technical challenge.

In any case, given the threat posed by malicious user actions – whether or not those actions be intentional – mitigation approaches will become much more widespread. By unifying these approaches with anti-exploitation technology, solutions will be able to apply multiple layers of protection to many attacks.

[79] This term originates from the OSI 7 layer model, of which the most abstract is Layer 7, or the Application Layer. Layer 8 would therefore be the end-user themselves.

Anti-exploitation is imperfect and should be a last resort

As a final note, we must unfortunately conclude that anti-exploitation technology is one of the most technically complex areas in which security controls can be placed.

There is an important security concept known as the "defender's dilemma". Simply put – an attacker or researcher only has to find one weak spot, whereas the defender must anticipate or mitigate all attacks in advance. Because anti-exploitation technology rests on the razor's edge between attack and defense, it is particularly prone to new advances in attack methodology, with attackers and defenders vying constantly for new advances. For this reason, anti-exploitation technology will always be brittle in the face of revolutionary new exploit methods.

With this in mind, although generic and built-in anti-exploitation technology is a valuable addition to Windows, end-users should ensure that they take a well-rounded approach to endpoint security. The best approach is to deploy products that are able to provide non-signature based protection at the host network interface as well as mitigation technologies such as "application firewalling" and API protection to counter the effects of malicious user action, locally running malware and other attacks that have somehow bypassed the previous layers of protection.

Note that it should not be expected that *any* solution provide protection in all cases – there will always be individual applications that can be successfully exploited, even under very difficult conditions, provided the vulnerability is suitable and the attacker is prepared to create exploits that are specific to given systems or given applications. The first aim of anti-exploitation systems, then, should be to eliminate generic attacks, the so called "cook book" attack patterns which allow for the creation of scalable, re-useable exploitation frameworks and rapid exploit development.

For the present, and foreseeable future, all anti-exploitation approaches currently available can be bypassed in lab conditions or proof of concept exploits. This should not be taken as a technology failure, since some of these systems are almost 100% effective against real-world exploits. However, it should be taken as a challenge, and provide an impetus for continued innovation and development in the field.

About the Author

Ben Nagy is a Senior Security Engineer at eEye Digital Security

Bibliography

3APA3A, offtopic, *Bypassing client application protection techniques*, 2004, online at http://www.security.nnov.ru/advisories/bypassing.asp?l=EN

Abadi, M, Erligsson, U, et al, *A Theory of Secure Control Flow*, 2005, available at http://research.microsoft.com/research/pubs/view.aspx?tr_id=867

Abadi, M, Erligsson, U, et al, *Control-Flow Integrity*, 2005, available at http://research.microsoft.com/research/pubs/view.aspx?tr_id=868

Andersen, S, Abella, V, *Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3*, 2004, http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

Anisimov, A, *Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass*, 2005, online at http://www.maxpatrol.com/defeating-xpsp2-heap-protection.pdf

Anon, *Once upon a free()*, 2001, Phrack 57 article 9, http://www.phrack.org/show.php?p=57&a=9

Baratloo, A, Tsai, T, Singh, N, *Transparent Run-Time Defense Against Stack Smashing Attacks*. In Proceedings of the USENIX Annual Technical Conference, 2000, online at http://www.research.avayalabs.com/project/libsafe/doc/usenix00/paper.html

Bray, B, *Compiler Security Checks In Depth*, 2002, available at http://msdn.microsoft.com/

Bray, B, *Security Improvements to the Whidbey Compiler*, 2003, in weblog ‘C++ Potential’, http://blogs.msdn.com/branbray/archive/2003/11/11/51012.aspx

Butler, J, Anon, Anon, *Bypassing 3rd Party Windows Buffer Overflow Protection*, Phrack 62, Article 5, 2004, available at http://www.phrack.org/show.php?p=62&a=5

Collins, R, Paging Extensions for the Pentium Pro Processor, 1996, online at http://www.rcollins.org/ddj/Jul96/Jul96.html

Conover, M, Horovitz, O, *Reliable Windows Heap Exploits*, 2004, presented at CanSecWest 2004, online at http://cansecwest.com/csw04/csw04-Oded+Connover.ppt

Conover, M, *XPSP2 Heap Exploitation.ppt*, available at http://www.cybertech.net/~sh0ksh0k/heap/

Cowan, C, Barringer, M, et al, *FormatGuard: Automatic Protection From printf Format String Vulnerabilities*, 2001, in proceedings of 10th USENIX Security Symposium, online at http://www.usenix.org/events/sec01/full_papers/cowanbarringer/cowanbarringer.pdf

Cowan, C, Beattie, S, et al, *PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities*, 2003, in proceedings of 12th USENIX Security Symposium, online at http://www.usenix.org/events/sec03/tech/full_papers/cowan/cowan.pdf

Cowan, C, et al, *StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks*. In USENIX Security Conference, January 1998. online at http://sherry.ifi.unizh.ch/cowan98stackguard.html

Designer, S, *StackPatch*, http://www.openwall.com/linux

Dik, C, posting to comp.security.unix, January 2, 1997

Durden, T, *Bypassing PaX ASLR protection*, Phrack 59, Article 9, 2002, available at http://www.phrack.org/show.php?p=59&a=9

Eclipse, S, *kill-bill*, 2005, http://www.phreedom.org/solar/exploits/msasn1-bitstring/

Eilam, E, *Reversing: Secrets of Reverse Engineering*, Wiley, 2005, ISBN: 0764574817

Etoh, H, *GCC extension for protecting applications from stack-smashing attacks*, http://www.research.ibm.com/trl/projects/security/ssp/, June 20, 2005

Etoh, H, *Stack Protection Systems: (propolice, StackGuard, XP SP2)*, 2005, Presented at CanSecWest 2005, online at http://cansecwest.com/core05/propolice-cansec2005.ppt

Ferrie, P, Perriot, F, *Mostly Harmless*, in Virus Bulletin, August 2004, online at http://pferrie.tripod.com/vb/sasser.pdf

Flake, H, *Third Generation Exploitation*, presented at BlackHat USA 2002, online at http://blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt

Funnywei, jerry, *Windows Xp Sp2*[illegible], available at http://www.xfocus.net/articles/200412/762.html

Gates, B, *Microsoft Progress Report: Security*, 2004, available at http://www.microsoft.com/mscorp/execmail/2004/03-31security.asp

Hardened Gentoo, http://www.gentoo.org/proj/en/hardened/

Hoglund, G, McGraw, G, *Exploiting Software: How to Break Code*, Addison Wesley Professional, 2004, ISBN: 0201786958

Hoglund, G, *The cross-page overwrite and it's application in heap overflows*, 2002, online at http://www.rootkit.com/vault/hoglund/The cross.pdf

Homepage of the PaX Team, http://pax.grsecurity.net/

Horovitz, O, *Big Loop Integer Protection*, 2002, Phrack 60 article 9, http://www.phrack.org/show.php?p=60&a=9

Huang, Y, *Protection Against Exploitation of Stack and Heap Overflows*, 2003, online at http://www.cgisecurity.com/lib/AntiOverflows.pdf

Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 3*, 2005, section 3.8, available at http://developer.intel.com/design/pentium4/manuals/index_new.htm

Intel Corporation, *Overview of the Protected Mode Operations of the Intel Architecture*, http://www.intel.com/design/intarch/papers/exc_ia.htm

Ionescu, A, *Introduction to NT Internals, Part 1*, 2004, http://www.relsoft.net/Articles/Process/part1.pdf

Kaempf, M, *Vudo - An object superstitiously believed to embody magical powers*, 2001, Phrack 57, article 8, http://www.phrack.org/show.php?p=57&a=8

Kiriansky, V, Breuning, D, Amarasinghe, S, *Secure execution via program shepherding*, in 11th USENIX Security Symposium, 2002, online at http://www.cag.lcs.mit.edu/commit/papers/02/RIO-security-usenix.pdf

Klog, *Frame Pointer Overwrite*, 1999, http://www.phrack.org/show.php?p=55&a=8

Koziol, J, Litchfield, D, et al, *The Shellcoder's Handbook : Discovering and Exploiting Security Holes*, John Wiley & Sons, 2004, ISBN: 0764544683

Litchfield, D, *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server*, 2003, http://www.nextgenss.com/papers/defeating-w2k3-stack-protection.pdf

Litchfield, D, *Windows Heap Overflows*, presented at BlackHat USA 2004, online at http://blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

LSD, *Win32 Assembly Components*, 2002, http://www.lsd-pl.net/documents/winasm-1.0.1.pdf

McDonald, A, *FAR (Function Address Retrieving)*, in blog arnold.mcdonald, http://arnold.mcdonald.free.fr/php/Main.php?p=1007, 2004, accessed June 20, 2005

MITRE, *CVE - Common Vulnerabilities and Exposures*, http://cve.mitre.org/

Moore, B, *Shattering By Example*, 2003, online at http://www.security-assessment.com/Whitepapers/Shattering_By_Example-V1_03102003.pdf

One, A, *Smashing The Stack For Fun And Profit*, Phrack 49, Article 14, 1996, online at http://www.phrack.org/show.php?p=49&a=14

Pageexec, *PaX privilege elevation security bug, posting to bugtraq mailing list*, March 5, 2005, http://www.securityfocus.com/archive/1/392348/2005-03-05/2005-03-11/0

PaX Team, *PAGEEXEC Documentation*, http://pax.grsecurity.net/docs/pageexec.txt

Pietrek, M, *A Crash Course on the Depths of Win32™ Structured Exception Handling*, in Microsoft Systems Journal, January 1997, online at http://www.microsoft.com/msj/0197/exception/exception.aspx

Pincus, J, Baker, B, *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*, in IEEE Security & Privacy, July / August 2004, online at http://research.microsoft.com/users/jpincus/beyond-stack-smashing.pdf

Pincus, J, Baker, B, *Mitigations for Low-Level Coding Vulnerabilities: Incomparability and Limitations*, 2004, online at http://research.microsoft.com/users/jpincus/mitigations.pdf

Provos, N, *Systrace - Interactive Policy Generation for System Calls*, 2003, homepage at http://www.citi.umich.edu/u/provos/systrace/

Rattle, *Using Process Infection to Bypass Windows Software Firewalls*, Phrack 62, Article 13, available at http://www.phrack.org/show.php?p=62&a=13

Ren, C, Weber, M, McGraw, G, *Microsoft Compiler Flaw Technical Note*, 2002, online at http://www.cigital.com/news/mscompiler-tech.pdf

Richarte, G, *Four different tricks to bypass StackShield and StackGuard protection*, 2002, http://www1.corest.com/files/files/11/StackguardPaper.pdf

Roberston, W, Kruegel, C, et al, *Run-time Detection of Heap-based Overflows*, 2003, in proceedings of 17th Large Installation Systems Administration Conference (LISA), online at http://www.usenix.org/publications/library/proceedings/lisa03/tech/full_papers/robertson/robertson.pdf

Shacham, H, Page, M, et al, *On the effectiveness of address-space randomization*, in Proceedings of the 11th ACM conference on Computer and communications security, 2004 online at http://www.stanford.edu/~blp/papers/asrandom.pdf

Silberman, P, Johnson, R, *A Comparison of Buffer Overflow Prevention Implementations and Weaknesses*, 2004, available at http://blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf

Singh, A, *Securing Memory*, 2004, in blog kernelthread.com, online at http://www.kernelthread.com/publications/security/smemory.html

Soeder, D, Permeh, R, Ukai, Y, *Advanced Return Address Discovery using Context-Aware Machine Code Emulation*, presented at Blackhat 2004, available at http://blackhat.com/presentations/bh-usa-04/bh-us-04-soeder/bh-us-04-soeder-up.ppt

Tuck, N, Calder, B, Varghese, G, *Hardware and Binary Modification Support for Code Pointer Protection From Buffer Overflow*, in Proceedings of the 37th International Symposium on Microarchitecture, December, 2004, online at http://www-cse.ucsd.edu/~calder/papers/MICRO-04-CodePointerProtection.pdf

Vendicator, *Stack Shield*, http://www.angelfire.com/sk/stackshield/, June 20, 2005

Wikipedia Entry, *Control Flow Graph*, 2005, http://en.wikipedia.org/wiki/Control_flow_graph, accessed June 20, 2005

Wilander, J, Kamkar, M, *A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention*, in Proceedings of the 10th Network and Distributed System Security Symposium, 2003, online at http://www.ida.liu.se/~johwi/research_publications/paper_ndss2003_john_wilander.pdf

Wojtczuk, R, *Defeating Solar Designer non-executable stack patch*, 1998, posting to bugtraq mailing list, online at http://www.insecure.org/sploits/non-executable.stack.problems.html

Acknowledgements

All the eEye family, in particular Derek, from whom stems any accuracy contained herein – how many proofreaders notice mistyped x86 opcodes? Other greets to BKK security farang, the real so1o, caddis and the excellent staff@ruxcon, and Khon Kaen Breweries, Ltd.