Address-Space Randomization for Windows Systems*

Lixin Li and James E. Just
Global InfoTek, Inc., Reston, VA
{nli,jjust}@globalinfotek.com

R. Sekar
Stony Brook University, Stony Brook, NY
sekar@cs.stonybrook.edu

Abstract

Address-space randomization (ASR) is a promising solution to defend against memory corruption attacks that have contributed to about three-quarters of US-CERT advisories in the past few years. Several techniques have been proposed for implementing ASR on Linux, but its application to Microsoft Windows, the largest monoculture on the Internet, has not received as much attention. We address this problem in this paper and describe a solution that provides about 15-bits of randomness in the locations of all (code or data) objects. Our randomization is applicable to all processes on a Windows box, including all core system services, as well as applications such as web browsers, office applications, and so on. Our solution has been deployed continuously for about a year on a desktop system used daily, and is robust enough for production use.

1 Introduction

An overwhelming majority of security advisories from US CERT in recent years has been attributed to memory corruption attacks. Typically, these attacks enable a remote attacker to execute arbitrary code on the victim system, thereby providing a mechanism for self-propagating worms, installation of backdoors (including "bot" software), spyware, or rootkits. Address-space randomization (ASR) [16, 3, 23, 4] provides a general defense against memory corruption attacks. Although several ASR techniques have been described for Linux [16, 3, 23, 4], to the best of our knowledge, there hasn't been any previous work describing ASR for the largest monoculture on the Internet, namely, the Microsoft Windows platform. We address this problem, and describe a system called DAWSON ("Diversity Algorithms for Worrisome SOftware and Networks") that provides about 15-bits of randomness in the locations of all code or data.

In parallel with our work, some commercial products for Windows ASR have begun to emerge, namely, Wehntrust [21] and Ozone [20]. In addition, Windows Vista is going to be shipped with a limited implementation of ASR [9]. However, these products suffer from one or more of the following drawbacks:

- *Insufficient range of randomization.* Windows Vista randomizes base addresses over a range of 256 possible values. This level of randomization is hardly sufficient to defeat targeted attacks: the attacker simply needs to try their attack an average of 128 times before succeeding. This isn't likely to significantly slow down self-replicating worms either. Wehntrust and Ozone provide more randomization, but significantly less than that of DAWSON in some memory regions such as the stack.
- *Incomplete randomization.* Often, only a subset of memory regions are randomized. For instance, Wehntrust does not randomize some memory regions. With Ozone, no information is available beyond the fact that the stack and the DLLs are randomized. Unfortunately, if the address of any writable memory region is predictable, the attacker can modify their attack so as to inject code into this region and execute it. Therefore, DAWSON randomizes all such memory regions.
- *Lack of detailed analysis.* With Wehntrust and Ozone, even the most basic information about their implementation (e.g., the complete list of memory regions that are randomized) isn't available. As a result, one cannot independently analyze or evaluate the quality of protection provided by them. In contrast, we provide a detailed analytical as well as experimental evaluation of DAWSON.

1.1 Contributions of this paper

- ***Development of practical techniques for realizing ASR on Windows.*** The architecture of Windows is quite different from UNIX, and poses several unique challenges that necessitate the development of new techniques for realizing randomization. Some of these challenges are:
  - *Difficulty of relocating critical DLLs.* Security-critical DLLs such as `ntdll` and `kernel32` are mapped to a fixed memory location by Windows very early in the boot process. Since most of the APIs targeted by attack code, including all of the system calls, reside in these DLLs, we needed to

*This work was partially funded by Defense Advanced Research Project Agency under contract FA8750-04-C-0244. Sekar's work was also supported in part by an ONR grant N000140110967 and NSF grants CNS-0208877 and CCR-0205376. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Project Agency, NSF, ONR, or the U.S. Government.

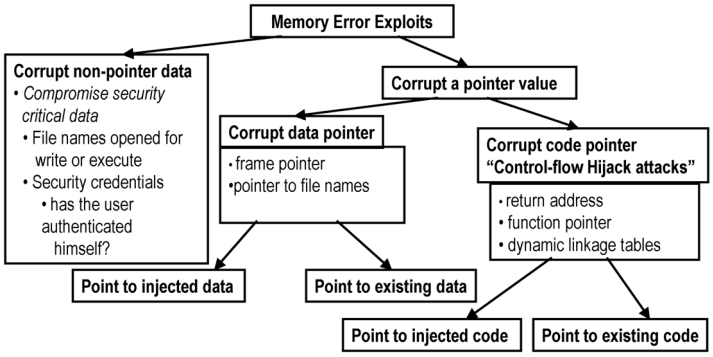

Figure 1. Memory Error Exploits.

develop techniques to relocate these DLLs.

– *Lack of access to OS or application source code.* This means that the primary approach used by Linux ASR implementations, namely, modification of the kernel source, is not an option on Windows.

– *Lack of UNIX-style shared libraries.* In UNIX, dynamically loaded libraries contain position-independent code, which allows them to be shared even if they are loaded at different addresses by different processes. In contrast, Windows DLLs need to be stored at the same memory address by all processes that share a single copy of the DLL.

- ***Robust implementation.*** DAWSON applies ASR to all Windows services, as well as user applications. We have had this system running on a production laptop installed with Windows XP, Office, and SQL and IIS servers. In addition to the standard set of OS services and applications, we have routinely used Office applications, Windows development tools (MSVC, .Net, etc.), SQL and IIS servers, and web browsers (e.g., IE and Firefox), and haven't experienced any compatibility or robustness issues. Performance overhead of DAWSON is typically under 5%.
- ***Analytical evaluation of effectiveness.*** We develop a classification of memory corruption attacks, and use it subsequently to compare previous works. We then provide a detailed analytical evaluation of our approach and provide estimates of success probabilities for various types of attacks.
- ***Experimental evaluation.*** We have evaluated the ability of DAWSON to defeat memory corruption attacks using 5 real-world exploits, as well as a range of sophisticated memory corruption attacks on a synthetic application.

1.2 Paper Organization

We begin with a classification of memory corruption attacks in Section 2 and describe the strengths and weaknesses of ASR in defending against these attacks. With this background, we then provide a comparison of DAWSON with previous work in Section 3. Our technical approach is described in Section 4, followed by analytical and experimental evaluation in Sections 5 and 6 respectively. Finally, concluding remarks appear in Section 7.

2 ASR and Memory Corruption Attacks

Intuitively, a memory error occurs in C programs when the object accessed via a pointer expression is different from its *referent*, i.e., the object intended by the programmer. Memory errors are classified into *spatial* and *temporal* errors. Temporal errors occur when a pointer is dereferenced beyond the lifetime of its referent, e.g., access to freed (or reallocated) memory. Security exploits haven't targeted temporal errors, and indeed, it is unclear how they can be used in attacks. Consequently, security research has focused on *spatial errors,* which are caused by (a) dereferencing a pointer that holds a value outside of the bounds of its referent, or (b) dereferencing a non-pointer or a corrupted pointer value.

Figure 1 classifies spatial memory error exploits based on whether pointer corruption is involved or not; and if so, whether the corrupted pointer refers to a data or code object; and if this object holds values provided ("injected") by the attacker or values already existing in victim process memory. Within "pointer corruption attacks," we include attacks that corrupt values used in address computation, e.g., an integer value used as an array subscript.

Absolute address randomization (AAR), used in [16, 3, 23], randomizes the absolute memory address of various code and data objects, but the relative distances between objects aren't randomized. AAR blocks pointer corruption attacks, since the attacker is no longer able to predict the object that will be referenced by a corrupted pointer value. For instance, a stack-smashing attack involves overwriting the return address on the stack with a value that points to a buffer variable used to hold attacker-provided data. The lo-

cation of this variable becomes unpredictable in the presence of AAR, thus defeating the attack.

Relative address randomization (RAR) techniques [4] randomize inter-object distances as well, and hence can defeat non-pointer attacks. DAWSON implements only AAR, and hence addresses only the pointer-corruption attacks (4 of the 5 categories shown in Figure 1). In practice, AAR is very effective since memory error exploits reported so far have been based on pointer corruption.

2.1 Limitations of AAR

We list the known limitations of ASR here, referring the reader to previous works such as [3] for details.

- *Local attacks* are launched from the same host as the victim application. AAR is based on the secrecy of a "randomization key." In practice, however, it is hard to protect this secret from local users and applications, and hence AAR has been used mainly to defend against remote exploits. Indeed, most AAR implementations (including DAWSON) aren't even designed to defend against local threats, including threats due to malicious code that, by some mechanism, runs on the same host as the victim application.
- *Relative address attacks* don't rely on absolute locations of data. Examples include data corruption attacks that don't require pointer corruption, and partial pointer corruption attacks that overwrite only the least significant byte(s) of an address.
- *Information leakage attacks* utilize a vulnerability to first read the value of some pointer, and then use it to compute the location of other objects in memory[1].
- *Brute-force (guessing) attacks* repeatedly attempt to guess the value to be used for corrupting a pointer.
- *Double-pointer attacks* require two vulnerabilities that are exploited in two steps. First, an attacker picks a random memory address A, and writes attack code at this address by exploiting an absolute address vulnerability, e.g., a heap overflow or a format-string bug. In the second step, the attacker uses a relative address vulnerability (e.g., a buffer overflow) to overwrite a code pointer with A.

DAWSON does not defend against the first three of these attack types, but as discussed in Section 5, it provides probabilistic protection against the other two.

2.2 Need to Relocate All Memory Regions

In addition to these limitations, some implementations of AAR may suffer from the weakness that the locations of some memory objects may not be randomized. This limitation can totally undermine the effectiveness of AAR, as we describe below. If a code region S is not randomized, then the attacker can execute a return-to-existing code attack into S. Of particular relevance in Windows is the common use of the instruction sequence `jmp esp` which causes a control-transfer to the top of the stack. During attacks, it is common for the top of the stack to contain attacker provided data. Thus, this instruction sequence allows for execution of injected code.

Any unrandomized writable section W poses a major threat, as it is possible to mount a 2-step attack as follows. In the first step, the attacker injects a short opcode sequence (such as `jmp esp` or other sequence that can utilize values in registers) into W. In the next step, control is transferred to this code[2].

Unrandomized read-only data sections don't pose as great a threat. Note that until the attacker's code gets control, it is not possible to "read" the contents of arbitrary memory in order to obtain values of pointers etc. However, there is a small chance that the read-only region contains data that corresponds to an exploitable instruction sequence.

3 Related Work

We use Figure 1 to compare previous techniques for memory error exploit protection. Early techniques such as StackGuard [7] and RAD [5] focused on protecting return addresses. ProPolice [8] extends StackGuard to protect all data on the stack from buffer overflow attacks, but does not address attacks on heap or static data. Libsafe/Libverify [1] also targets stack-smashing vulnerabilities, but does so without requiring source-code access. [17] shows how to use binary-rewriting to implement RAD.

PaX [16], address obfuscation [3], and transparent runtime randomization [23] use memory layout randomization to defeat pointer corruption attacks on Linux. DAWSON achieves the same effect on Windows. Relative address randomization, in addition to absolute address randomization, is achieved in [4] using a source-to-source transformation.

Non-executable data segments and instruction set randomization [2, 13] address all injected code attacks. Program shepherding [14] uses runtime monitoring of branch targets to defeat injected code attacks.

Pointguard [6] randomizes ("encrypts") stored pointer values, and can hence be effective against all pointer corruption attacks. However, the approach does not consider the possibility that pointers may be

[1] Most previous approaches such as StackGuard (with random or XOR canary) [7], ProPolice [8] and PointGuard [6] are susceptible to such attacks as well.

[2] Such attacks require absolute-address vulnerabilities such as heap overflows or format-string bugs that are quite common.

Type	Description	Protection	Granularity of Rebasing
Free	Free space	Inaccessible	Not rebased
Code	Executable or DLL code	Read-only	15 bits
Static data	Within executable or DLL	Read-Write	15 bits
Stack	Process and thread stacks	Read-Write	29 bits
Heap	Main and other heaps	Read-Write	20 bits
TEB	Thread Environment Block	Read-Write	19 bits
PEB	Process Environment Block	Read-Write	19 bits
Parameters	Command-line and Environment variables	Read-Write	19 bits
VAD	Returned by virtual memory allocation routines	Read-Write	15 bits
VAD	Shared Info for kernel and user mode	Unwritable	Not rebased

Figure 2. Types of regions within virtual memory of a Windows process.

aliased with non-pointer data, and hence can break legitimate programs.

Complete memory error protection techniques can deterministically stop all memory exploits, but they impose a significant overhead [24, 11, 19] and/or suffer from incompatibility with legacy code [15, 10].

3.1 ASR Implementations on Windows

Wehntrust [21] is a third-party implementation of ASR for Windows. As compared to our technique, it does not randomize all memory regions — in particular, several writable memory regions, including the environment variables, arguments, and the PEB/TEB aren't relocated. (Their product literature does claim the ability to relocate PEB/TEB, but in version 1.0.0.9 of their software, PEB/TEB wasn't relocated.) In addition, they provide 19 bits or less randomness in the stack, as opposed to the 29 bits in our implementation.

Ozone is another third-party ASR implementation, but we haven't been able to find any information on their product other than [20], according to which they randomize the stack and the DLLs, but it is unknown if the heap and other regions of the process memory are randomized. Moreover, randomness in stack addresses is much smaller than ours — just 16-bits.

Windows Vista is going to incorporate a limited amount of ASR. According to [9], the stack, heap, DLLs and the executables that ship with the OS are randomized, but it is unclear whether other regions are. More importantly, they use only 8-bits of randomness, which makes brute-force attacks much easier than in DAWSON[3]. Moreover, older versions of the OS, including XP and 2003, are going to be widely deployed for a long time to come, and hence our solution would still be necessary, even if one were to be satisfied with 8-bits of randomness.

[3]Vista relies on a combination of NX (non-executable data), ASR and other techniques to defeat memory corruption attacks, and hence its designers seem to believe that 8-bit randomization is adequate.

4 Approach Description

We use the following techniques to implement AAR on Windows without modifying the kernel or applications:

- *Injecting a randomization DLL into a target process:* Much of the randomization functionality is implemented in a DLL. We ensure that this DLL gets loaded very early in the process creation. This DLL "hooks[4]" standard Windows API functions relating to memory allocation, and randomizes the base address of all memory regions.
- *Customized loader:* Some of the memory allocation happens prior to the time when the randomization DLL gets loaded. To randomize memory allocated prior to this point, we make use of a customized loader, which makes use of lower level API functions provided by `ntdll` to achieve randomization.
- *Kernel driver[5]:* Base addresses of some DLLs are determined very early in the boot process, and to randomize these, we have implemented a boot-time driver. In a couple of instances, we had to resort to in-memory patching of the kernel executable image, so that some hard-coded base addresses can be replaced by random values. (Naturally, such patching is kept to a bare minimum in order to minimize porting efforts across different versions of Windows.)

We use these techniques to randomize the base address of different memory regions in Windows as shown in Figure 2. Below, we describe our approach for randomizing each of these memory regions.

[4]"Hooking" is the term used in Windows literature to refer to interception of function calls, typically in DLLs. There are several standard techniques for this, and the interested reader is referred to [18, 12].

[5]The term "driver" in Windows literature corresponds roughly to the term "kernel module" in UNIX literature. In particular, it isn't necessary for such drivers to be associated with any devices.

4.1 Dynamically Linked Libraries

UNIX operating systems generally rely on shared libraries, which contain position-independent code. This means that they can be loaded anywhere in virtual memory, and no relocation of the code would ever be needed. This has an important advantage: different processes may map the same shared library at different virtual addresses, yet be able to share the same physical memory. In contrast, Windows DLLs contain absolute references to addresses within themselves, and hence are not position-independent. Specifically, if a DLL is to be loaded at a different address from its default location, then it has to be explicitly *rebased*, which involves updating absolute memory references within the DLL to correspond to the new base address.

Since rebasing modifies the code in a DLL, there is no way to share the same physical memory on Windows if two applications load the same DLL at different addresses. As a result, the common technique used in UNIX for library randomization, i.e., mapping each library to a random address as it is loaded, would be very expensive on Windows: it requires a unique copy of each library for every process. To avoid this, our approach is to rebase a library the first time it is loaded after a reboot. All processes will then share this same copy of the library[6]. This default behavior for a DLL can be changed by explicit configuration, using a DAWSON-specific entry in the Windows Registry.

Rebasing is implemented by hooking the `NtMapViewOfSection` function in `ntdll`, and changing a parameter that specifies the new base address. This approach does not work for certain libraries such as `ntdll` and `kernel32` that get loaded very early during the reboot process. We have developed a kernel-mode driver to rebase such DLLs. Specifically, we use an offline process to create a (randomly) rebased version of these libraries (using the standard rebase tool) before a reboot. Then, during the reboot, our custom boot-driver gets loaded before the Win32 subsystem is started up, and overwrites the disk image of these libraries with the corresponding rebased versions. When the Win32 subsystem starts up, these libraries are now loaded at random addresses.

Note that when the base of a DLL is randomized, the base address of code as well as static data within the DLL gets randomized. The granularity of randomization that can be achieved is somewhat coarse, since Windows requires DLLs to be aligned on a 64K boundary, thus removing 16-bits of randomness. In addition, applications can typically use only up to 2GB of memory on Windows, thus reducing the randomness in DLL addresses to 15-bits.

4.2 Stack Randomization

Unlike UNIX, where multithreaded servers aren't the norm, most servers on Windows are multi-threaded. Moreover, most request processing is done by child threads, and hence it is more important to protect the thread stacks. Our approach to randomize thread stacks is based on hooking the `CreateRemoteThread` call, which in turn is called by `CreateThread` to create a new thread. This routine takes the address of a start routine as a parameter, i.e., execution of the new thread will begin with this routine. We replace this parameter with the address of a "wrapper" function written by us. This wrapper function first allocates a new thread stack at a randomized address by hooking `NtAllocateVirtualMemory`. However, this isn't sufficient, since the allocated memory has to be aligned on a 4K boundary. Since only the lower 2GB of address space is typically usable, this leaves us with only 19-bits of randomness. To increase this, our wrapper routine decrements the stack by a random number between 0 and 4K that is a multiple of 4. (Stack should be aligned on a 4-byte boundary.) This provides an additional 10-bits of randomness (for a total of 29 bits).

The above approach does not work for randomizing the main thread that begins execution when a new process is created. This is because the `CreateThread` isn't involved in the the creation of this thread. To overcome this problem, we have written a "wrapper" program to start an application that is to be randomized. This wrapper is essentially a customized loader. It uses the low-level call `NtCreateProcess` to create a new process with no associated threads. Then the loader explicitly creates a thread to start executing in the new process, using a mechanism similar to the above for randomizing the thread stack. The only difference is that this requires the use of a lower-level function `NtCreateThread` rather than `CreateThread` or `CreateRemoteThread`.

4.3 Executable Base Address Randomization

In order to "rebase" the executable, we need the executable to contain *relocation* information. This information is normally included in DLLs but is not typically present in release versions of COTS binaries[7]. This requires a minimal level of cooperation from the

[6]This means that an attack based on the absolute location of a library may succeed against all application that use this library. However, since our main goal is to stop an attack before it compromises any one process, this limitation isn't a significant concern. But it does mean that the analytical results in Section 5 must be interpreted as the number of attempts across all processes running on a host since a reboot.

[7]This is true in the UNIX world as well — applications need to be compiled with relocation information or as position-independent code in order for ASR to be applicable.

software vendor. As ASR gradually gains acceptance, we believe that vendor cooperation will become easier to obtain. We point out that unlike debug information, there are no significant intellectual property concerns with providing relocation information.

When relocation information is available, rebasing of executables is similar to that of DLLs: an executable is rebased just before it is executed for the first time since a reboot, and future executions can share this same rebased version.

If relocation information is not present, then the executable cannot be rebased[8]. While randomization of other memory regions protects against most known exploits, an attacker can craft "return-to-exe" attacks that exploit the code already present in the executable.

4.4 Heap Randomization

Windows applications typically use many heaps, each created using a `RtlCreateHeap` function. We hook this function and modify the base address of the new heap. Due to alignment requirements, this can provide only 19 bits of randomness. To increase it further, we hook individual requests for allocating memory from this heap, specifically, `RtlAllocateHeap`, `RtlReAllocate`, and `RtlFreeHeap`, and increase allocations by 8 or 16 bytes, providing one more bit of randomness for a total of 20 bits.

The above approach is not applicable for rebasing the main heap, since its address is determined before the randomization DLL is loaded. Specifically, the main heap is created using a call to `RtlCreateHeap` within the `LdrpInitializeProcess` function. Our kernel driver patches this call and transfers control to a wrapper function. This wrapper function modifies a parameter to the `RtlCreateHeap` so that the main heap is rebased at a random address aligned on a 4K page boundary.

In addition, we add a 32-bit "magic number" to the headers used in heap blocks to provide additional protection against heap overflow attacks. Heap overflow attacks operate by overwriting control data used by heap management routines. This data resides next to the user data stored in a heap-allocated buffer, and hence could be overwritten using a buffer overflow vulnerability. Typically, the attack takes effect when the overwritten block is freed. By checking the magic-number at this point, we make it virtually impossible to carry out this type of attack.

[8]It isn't possible to simply analyze the binary to reconstruct relocation information. This is because there is no way to distinguish pointers (that need to be relocated) from integer values (which should not be relocated) in binary code: both would typically appear as constants in binary code.

4.5 PEB and TEB

PEB and TEB are created in kernel mode, specifically, in the `MiCreatePebOrTeb` function of `ntoskrnl.exe`. The function itself is complicated, but the algorithm for PEB/TEB location is simple: it searches the first available address space from an address specified in a variable `MmHighestUserAddress`. Our approach is to patch the memory image of `ntoskernel.exe` in our boot driver so that it uses the contents of an another variable `RandomizedUserAddress` that is initialized by our boot driver. By initializing this variable with different values, PEB and TEB can be located at any of 2^{19} possible 4K-aligned addresses within the first 2GB of memory.

4.6 Environment/Command-line parameters

On Windows, environment variables and process parameters reside in separate memory areas. In normal programs, they are accessed using a pointer stored in the PEB, but if their locations are predictable, then an attacker can use them directly in attacks. To relocate them, our approach is to allocate randomly-located memory and copy over the contents of the original environment block and process parameters to the new location. Following this, the original regions are marked as inaccessible, and the PEB field is updated to point to the new locations.

4.7 VAD Regions

There are two types of VAD regions [22]. The first type is normally at the top of user address space (on SP2 it is `0x7ffe0000-0x7ffef000`). These pages are updated from kernel and read by user code, thus providing processes with a faster way to obtain information that would otherwise be obtained using system calls. This type of pages are created in the kernel mode and are marked read-only, and hence we don't randomize their locations. A second type of VAD region represents actual virtual memory allocated to a process using `VirtualAlloc`. For these regions, we wrap the `VirtualAlloc` function and modify its parameter `lpAddress` to a random multiple of 64K.

4.8 Discussion

Attacks on DAWSON Implementation. In addition to attacks on randomization, which have previously been discussed, there could be attacks on DAWSON implementation and the runtime infrastructure used by it. It is important to note that DAWSON is targeted at protecting *benign applications* from *remote exploits*. Malicious applications could try to subvert the API-hooking mechanism used in DAWSON implementation, but benign applications won't exhibit such

behavior until such time malicious code is injected into its memory and starts execution. However, DAWSON will prevent execution of such code, and hence subversion of API-hooking is not a real threat to DAWSON.

Local attacks, such as those launched by malicious code that may already be resident on the victim system, aren't our focus. Local attacks can indeed subvert or disable the entire DAWSON system. As mentioned before, previous AAR techniques such as those of [16, 3, 23] are also defeated by local attacks since they don't provide mechanisms to protect randomization keys from local processes.

Memory Fragmentation One commonly cited problem of ASR is that of possible memory fragmentation, which may significantly reduce *usable memory space* for applications. We point out, however, that there is a simple way to avoid memory fragmentation: generate a random key k, and add this to the default location at which each memory object would have been allocated in the absence of ASR. Addresses are "wrapped" around as needed to stay within user-addressable memory, and the lower-order bits zeroed out to meet alignment requirements. The value of k can be different for different processes in UNIX, but has to be shared across all processes in Windows due to its use of DLLs as opposed to shared libraries. Naturally, the technique can be extended to use different keys for different memory regions such as DLLs, stacks and heaps, but this will introduce some amount of fragmentation. In general, the trade-off is between the number of different random keys used and the degree of fragmentation.

Portability across Windows versions. DAWSON is primarily based on hooking several Win32 API functions and very few native API functions. Both these APIs are quite stable (with respect to the functionalities relevant to our implementation) across Windows 2000, XP, and 2003, thereby easing porting. In terms of porting efforts, the main effort has been in PEB/TEB porting, since it relies on kernel patching. It took us several hours to port the patch from Windows SP1 to SP2, and Windows 2003. Other parts of implementation, which were based on user level DLLs and kernel mode drivers, did not require any change.

5 Analytical Evaluation

In this section, we estimate the effort needed to defeat attack classes that specifically target our approach.

5.1 Brute-Force Attacks

Figure 3 summarizes the expected number of attempts required for different attack types. Note that the expected number of attacks is given by $1/2p$, where p is the success probability for an attack. The numbers

Attack type	Attack target	
	Stack/Heap	Static data/code
Injected code	262K*	16.4K
Existing code	N/A	16.4K
Injected data	262K*	16.4K
Existing data	$> 524K$	16.4K

Figure 3. Expected attack attempts.

marked with an asterisk depend on the size of the attack buffer, and a value of 4KB has been assumed to compute the numbers in the table.

Note that an increase in number of attack attempts translates to a proportionate increase is the total amount of network traffic to be sent to a victim host before expecting to succeed. For instance, the expected amount of data to be sent for injected code attacks on stack is $262K * 4K$, or about 1GB. For injected code attacks involving buffers in the static area, assuming a minimum size of 128 bytes for each attack request, is $16.4K * 128 = 2.1MB$.

Injected code attacks. For such attacks, note that the attacker has to first send malicious data that gets stored in a victim program's buffer, and then overwrite a code pointer with the absolute memory location of this buffer. Our approach does not disrupt the first step, but foils the second step with a high probability. The probability of a correct guess can be estimated from the randomness in the base address of different memory regions:

- *Stack:* Figure 2 shows that there is 29 bits of randomness on stack addresses, thus yielding a probability of $1/2^{29}$. To increase the odds of success, the attacker can prepend a long sequence of NOPs to the attack code. A NOP-padding of size 2^n would enable a successful attack as long as the guessed address falls anywhere within the padding. Since there are 2^{n-2} possible 4-byte aligned addresses within a padding of 2^n-bytes, the success probability becomes $1/2^{31-n}$.
- *Heap:* Figure 2 shows that there is 20 bits of randomness in heap addresses. Specifically, bits 3 and bits 13–31 have random values. Since a NOP padding of 4K bytes will only affect bits 1 through 12 of addresses, bits 13–31 will continue to be random. As a result, the probability of successful attack remains $1/2^{19}$ for a 4K padding. It can be shown that for larger NOP padding of 2^n bytes, the probability of successful attack remains $1/2^{31-n}$.
- *Static data:* According to Figure 2, there are 15-bits of randomness in static data addresses: specifically, the MSbit and the 16 LSbits aren't random. Since the use of NOP padding can only address randomness in the lower order bits of address that are already pre-

CVE Id	Target	Attack Type	Effective?
CVE-2003-0533	Microsoft LSASS	Stack smash/code injection	Yes
CVE-2003-0818	Microsoft ASN.1 Library	Heap overflow/code injection	Yes
CVE-2002-0649	MSSQL 2000/MSDE	Stack smash/code injection	Yes
CVE-2002-1123	MSSQL 2000/MSDE	Stack smash/code injection	Yes
CVE-2003-0352	Microsoft RPC DCOM	Stack-smash/jump to EXE code	No

Figure 4. Effectiveness in stopping real-world attacks.

dictable, the probability of successful attacks remains $1/2^{15}$. (This assumes a NOP padding $< 64K$.)

Existing code attacks. An existing code attack may target code in DLLs or in the executable. In either case, Figure 2 shows that there are 15-bits of randomness in these addresses. Thus, the probability of guessing the desired code address is $1/2^{15}$.

Note that exploitable code sequences may occur at multiple locations within a DLL or executable, and this may increase the probability of successful attacks. However, note that the randomness in code addresses arise from all but the MSbit and the 16 LSbits. It is quite likely that different exploitable code sequences will differ in the 16 LSbits, which means that exploiting each one of them will require a different attack attempt. Thus, the probability of $1/2^{15}$ will likely hold.

Injected data attacks involving pointer corruption. The calculations and the results here are similar to that for injected code attacks[9].

Existing Data Attacks involving pointer corruption. The main difference between injected data and existing data attacks is that the idea of repeating the attack data isn't useful here. Thus, the probability of a successful attack on the stack is 2^{-29}, on the heap is 2^{-20} and on static data is 2^{-15}.

5.2 Double-pointer attacks

From an attacker's perspective, a double-pointer attack has the drawback that it requires two distinct vulnerabilities: an absolute address vulnerability and a relative address vulnerability. Its benefit is that the attacker need only guess a writable memory location, which requires far fewer attempts. For instance, if a program uses 200MB of data (10% of the roughly 2GB virtual memory available), then the likelihood of a correct guess for A is 0.1. For processes that use much smaller amount of data, say, 10MB, the success probability falls to 0.005.

[9]Note that the idea of NOP padding is applicable at a higher level for data attacks: replicate the attack data several times in order to account for some uncertainty in the location of the target object.

6 Experimental Evaluation

6.1 Functionality

We have implemented DAWSON on Windows 2003 and Windows XP platforms, including SP1 (build 2600, xpsp1.020828-1920) and SP2 (build 2600.xpsp2_rtm.040803-2158). Most tests have been done on XP versions with default configurations, and Microsoft Office 2003 and SQLServer 8.00.194.

Over the past year, we have been using a DAWSON-protected system as one of our development machines. We have routinely used applications such as the Internet Explorer, SQLServer, Windbg, Windows Explorer, Word, WordPad, Notepad, Regedit, and so on. We used `Windbg` to print the memory map of these applications and verified that all regions have been rebased to random addresses. The addition of randomization has been without a glitch, and has not caused any perceptible loss of functionality or performance.

6.2 Effectiveness on Real-world Attacks

We tested the effectiveness of DAWSON in stopping several real-world attacks. We used the Metasploit framework (`http://www.metasploit.com/`) for testing purposes. Our testing included all working metasploit attacks on Metasploit Version 2.4 that were applicable to our test platform (Windows XP SP1), and are shown in Figure 4. We first ran the exploits with DAWSON protections enabled but with randomization set to zero (i.e., no memory address is randomized) and verified that the exploits were successful. We then used DAWSON with non-zero randomization and verified that four of the five failed. The successful attack was one that relied on predictability of code addresses in the executable, since DAWSON could not randomize these addresses due to unavailability of relocation information for the executable section for this server. Had the EXE section been randomized, this attack would have failed as well[10].

[10]Specifically, it used a stack-smashing vulnerability to return to a specific location in the executable. This location had two `pop` instructions followed by a `ret` instruction. At the point of return, the stack top contained a pointer that pointed into a buffer on the stack that held the input from the attacker. This meant that the return instruction transferred control to the attacker's code stored in this buffer.

Program	Workload	Base Runtime	DAWSON Overhead	Standard Deviation
Notepad	Start up, open a 1.4 MB text file	1.031s	3.4%	2.5%
Winword	Start up, open a 42 MB word document	5.489s	3.2%	3.8%
Excel	Start up, open a 398KB spreadsheet	0.794s	2.9%	2.6%
Powerpoint	Start up, open a 4MB powerpoint file	1.216s	2.1%	2.2%
Sqlserver	Startup, login to database, run 5 SQL queries, shutdown	0.992s	3.1%	2.4%
Firefox	Start up and visit `www.google.com`	1.070s	10.5%	1.2%
Testheap	issue 1M malloc's of random-size blocks ranging up to 64K	9.395s	12.4%	1.9%

Figure 5. Performance overhead of DAWSON

6.3 Effectiveness on Sophisticated Attacks

The problem with real-world attacks is that they tend to be rather simple. In order to test the effectiveness against many different types of vulnerabilities, we developed a synthetic application that was seeded with several vulnerabilities. We then developed 14 distinct attacks to exploit these vulnerabilities:

- *stack buffer overflow* attacks that overwrite
  - *return address* to point to
    - 1. injected code on stack
    - existing `call esp` code in
      - 2. the executable
      - 3. `ntdll` DLL
      - 4. `kernel32` DLL
      - 5. one of the application's DLLs
    - 6. existing code in a DLL ("return-to-libc")
  - 7. a local function pointer to point to injected code
- *heap overflow* attacks that overwrite
  - 8. a local function pointer with address of DLL
  - 9. a function pointer in the PEB (`RtlCriticalSection` field) with DLL code address
- 10. a *heap lookaside list overflow* that overwrites the return address on the stack to point to DLL code
- 11. a *process heap critical section list overflow* that overwrites a function pointer with DLL address
- *integer overflow* attacks that overwrite
  - 12. a global function pointer with DLL address
  - 13. an exception handler pointer stored on the stack so that it points to existing code in a DLL
- 14. a *format string* exploit on a `sprintf` function that prints to a stack-allocated buffer.

We verified that when DAWSON is run with zero randomization, all these exploits worked on Windows XP SP1 as well as SP2. Finally, we ran DAWSON in normal mode and verified that all 14 attacks failed.

6.4 Runtime performance

Performance measurements were carried out on a Dell PowerEdge SC420 (2.8GHz Pentium 4 CPU with 2.5GB memory) running Microsoft Windows XP SP2.

Most of DAWSON overhead occurs at application initialization time. This is because during startup, operations that are associated with significant DAWSON overheads occur far more frequently than during steady-state operation, e.g., DLL rebasing, dynamic memory allocations and thread creations. For this reason, our measurements are concerned mainly with startup times. The overheads that we measured for various programs are shown in Figure 5. "Base runtime" refers to the total CPU time (in seconds) for running a benchmark without DAWSON. The numbers reported are the average across ten runs. Note that for most applications, the measured overheads were around 3%. Since this number is close to the standard deviation in our measurements, the overhead would essentially be imperceptible to a user. Firefox is an exception, and our analysis found that it performs a very large number of memory allocations at startup time (about 300K). Since DAWSON introduces significant overheads for `malloc` calls, Firefox startup is slowed down by 10%. When `malloc` randomizations were disabled, the overhead fell down to about 1%. This result is similar to that of `testheap`, a malloc-intensive micro benchmark we created.

In addition, we measured boot-time overheads, which is mainly concerned with creating rebased versions of `ntdll`, `kernel32` and `user32` DLLs on the disk. DAWSON added 0.53 seconds to the boot time, with a standard deviation of 3.2% across six runs.

7 Conclusion

In this paper, we presented DAWSON, a lightweight approach for effective defense of Windows-based systems against remotely launched memory corruption attacks. DAWSON protects all services and applications by randomizing their memory layout. Specifically, all code sections and writable data segments are rebased, providing a minimum of 15-bits of randomness in their locations. Our technique does not require access to the source code of applications or the operating system. However, in order to provide full protection, it does require a minimal level of help from the vendors in terms

of providing relocation information for the executables.

We established the effectiveness of DAWSON using a combination of theoretical analysis and experiments. DAWSON introduces low performance overheads, and does not impact the functionality or usability of protected systems. These factors make it a practical solution for stopping a broad range of memory corruption attacks. A widespread deployment of DAWSON can significantly alleviate the common mode failure problem for the Windows monoculture.

Acknowledgements

We would like to thank Karl Levitt and Jeff Rowe for numerous dicussions on the DAWSON project; Tufan Demir for his contributions to proof-of-concept prototypes for some of the techniques that were implemented into DAWSON; and Jason Li for developing an early version of our vulnerable synthetic application. We would also like to thank Mark Cornwell for his extensive contributions in testing and evaluation; and Jason Minto for his help in various phases of this project. Finally, we would like to thank Sandeep Bhatkar and the anonymous reviewers for their thoughtful reviews that significantly improved the final version of this paper.

References

[1] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Technical Conference*, 2000.

[2] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM Computer and Communications Security (CCS)*, 2003.

[3] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, 2003.

[4] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[5] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *IEEE Int'l Conference on Distributed Computing Systems*, 2001.

[6] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2003.

[7] Crispin Cowan et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[8] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web at URL `http://www.trl.ibm.com/projects/security/ssp/main.html`, 2000.

[9] Michael Howard. ASLR features in Windows Vista. Published on World-Wide Web at URL `http://blogs.msdn.com/michael_howard/archive/2006/05/26/608315.aspx`, 2006.

[10] Trevor Jim, Greg Morrisett, Dan Grossman, Micheal Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[11] Robert W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1997.

[12] Yariv Kaplan. API spying techniques for windows 9x, NT and 2000. Published on World-Wide Web at URL `www.internals.com/articles/apispy/apispy.htm`, 2000.

[13] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM Computer and Communications Security (CCS)*, 2003.

[14] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, 2002.

[15] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2002.

[16] PaX. Published on World-Wide Web at URL `http://pax.grsecurity.net`, 2001.

[17] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX Annual Technical Conference*, 2003.

[18] Mark Russinovich and Bryce Cogswell. Windows NT system-call hooking. *Dr. Dobb's Journal*, Jan 1997.

[19] Olatunji Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium*, 2004.

[20] Eugune Tsyrklevich. Ozone. Published on World-Wide Web `http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-tsyrklevich.pdf`, 2005.

[21] WehnTrust. Published on World-Wide Web at URL `http://www.wehnus.com/products.pl`, 2006.

[22] windbg. Published on World-Wide Web at URL `http://www.microsoft.com/whdc/devtools/debugging/installx86.mspx`, 2006.

[23] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *Symposium on Reliable and Distributed Systems (SRDS)*, 2003.

[24] Wei Xu, Daniel C. Duvarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2004.