Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits

Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar
Department of Computer Science,
Stony Brook University, Stony Brook, NY 11794
{sbhatkar,dand,sekar}@cs.sunysb.edu

Abstract

Attacks which exploit memory programming errors (such as buffer overflows) are one of today's most serious security threats. These attacks require an attacker to have an in-depth understanding of the internal details of a victim program, including the locations of critical data and/or code. *Program obfuscation* is a general technique for securing programs by making it difficult for attackers to acquire such a detailed understanding. This paper develops a systematic study of a particular kind of obfuscation called *address obfuscation* that randomizes the location of victim program data and code. We discuss different implementation strategies to randomize the absolute locations of data and code, as well as relative distances between data locations. We then present our implementation that transforms object files and executables at link-time and load-time. It requires no changes to the OS kernel or compilers, and can be applied to individual applications without affecting the rest of the system. It can be implemented with low runtime overheads. Address obfuscation can reduce the probability of successful attacks to be as low as a small fraction of a percent for most memory-error related attacks. Moreover, the randomization ensures that an attack that succeeds against one victim will likely not succeed against another victim, or even for a second time against the same victim. Each failed attempt will typically crash the victim program, thereby making it easy to detect attack attempts. These aspects make it particularly effective against large-scale attacks such as Code Red, since each infection attempt requires significantly more resources, thereby slowing down the propagation rate of such attacks.

1 Introduction

The C and C++ languages are popular primarily because of the precise low-level control they provide over system resources, including memory. Unfortunately, this control is more than most programmers can handle, as evidenced by the host of memory-related programming errors which plague software written in these languages, and continue to be discovered every day. Attacks which exploit memory errors such as buffer overflows constitute the largest class of attacks reported by organizations such as the CERT Coordination Center, and pose a serious threat to the computing infrastructure.

To date, a number of attacks which exploit memory errors have been developed. The earliest of these to achieve widespread popularity was the *stack smashing* attack [31, 27], in which a stack-allocated buffer is intentionally overflowed so that a return address stored on the stack is overwritten with the address of injected malicious code. (See Figure 1). To thwart such attacks, several approaches were developed, which, in one way or another, prevent undetected modifications to a function's return address. They include the StackGuard [11] approach of putting *canary values* around the return address, so that stack smashing can be detected when the canary value is clobbered; saving a second copy of return address elsewhere [9, 6]; and others [16].

The difficulty with the above approaches is that while they are effective against stack-smashing attacks, they can be defeated by attacks that modify code pointers in the static or heap area. In addition, attacks where control flow is not changed, but security-critical data such as an argument to `chmod` or `execve` system call are changed, are not addressed. Recently, several new classes of vulnerabilities such as the integer overflow vulnerability (reported in Snort [34] and earlier in sshd [35]), heap overflows [23] and double-free vulnerabilities [2] have emerged. These developments lead us to conclude that additional ways to exploit the lack of memory safety in C/C++ programs will continue to be discovered in the future. Thus, it is important to develop approaches that provide systematic protection against all foreseeable memory error exploitations.

As a first step towards developing more comprehensive solutions against memory exploits, we observe that such exploits require an attacker to possess a detailed understanding of the victim program, and have precise knowl-

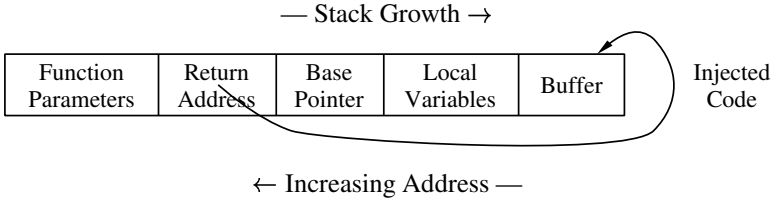

Figure 1: A buffer overflow in which the current function's return address is replaced with a pointer to injected code.

edge of the organization of data and code within the victim program memory. *Code obfuscation* is a general technique that attempts to secure programs by making them hard to understand. It is typically implemented using a set of randomized, semantics-preserving program transformations [38, 10, 4]. While code obfuscation is concerned primarily with preventing the understanding and reverse engineering of binary code, our interest lies in obfuscations which modify the internal runtime behavior of programs in ways that don't affect the observable semantics, but do create unpredictability which makes it difficult to successfully craft attacks which exploit memory errors.

Forrest, et.al. [17] suggested the use of randomized program transformations as a way to introduce *diversity* into applications. Such diversity makes it necessary for attackers to analyze each copy of the victim program independently, thereby greatly increasing the cost of developing attacks. They presented a prototype implementation that performed one particular kind of randomization: the randomization of the addresses of stack-resident data. Their implementation modified the gcc compiler to insert a random amount of padding into each stack frame. Our paper extends this basic idea, and presents a systematic study of the range of address randomizations that can be achieved using program transformation.

Address obfuscation is a program transformation technique in which a program's code is modified so that each time the transformed code is executed, the virtual addresses of the code and data of the program are randomized. As we will show, this makes the effect of most memory-error exploits non-deterministic, with only a very small chance of success. Attackers are forced to make many attempts on average before an attack succeeds, with each unsuccessful attack causing the target program to crash, increasing the likelihood that the attack will be detected. Moreover, an attack that succeeds against one victim will not succeed against another victim, or even for a second time against the same victim. This aspect makes it particularly effective against large-scale attacks such as Code Red, since each infection attempt requires significantly more resources, thereby greatly reducing the propagation rate of such attacks.

The PaX project has also developed an approach for randomizing the memory regions occupied by program code and data, called Address Space Layout Randomization (ASLR) (See `http://pageexec.virtualave.net` for documentation on PaX project.) Rather than viewing address obfuscation as a program transformation, they view it as an operating system feature. In particular, they have modified the Linux kernel so that it randomizes the base address of different sections of memory, such as the stack, heap, code, and memory-mapped segments. A key benefit of this approach is that it requires no changes to individual applications (other than having the compiler generate position-independent code). However, since the approach incorporates no analysis of the applications, it is difficult to perform address randomizations beyond changes to the base addresses of different memory segments. In contrast, a program transformation approach will permit randomization of the locations of individual variables and routines within these memory sections. Such randomization makes it difficult to carry out attacks that rely on relative distances between variables to modify critical data, e.g., a string used as an argument to `execve`. Moreover, it introduces significant additional diversity into the program, as it is no longer possible to craft attacks by knowing just the offsets in the base address of various memory segments. (These offsets can potentially be learned by exploiting vulnerabilities that may allow attackers to read contents of victim program memory without crashing it.)

The current generation of compilers and application binary interfaces limit how much randomization is possible, and at what stage (compile-time, link-time, load-time or runtime) such randomization can be performed. Our implementation focuses on techniques that can be smoothly integrated into existing OS environments. The key contribution of this paper is to develop and analyze the range of address obfuscations that can be implemented effectively with low runtime overheads. The principal benefits of this approach are:

- It systematically protects against a wide range of at-

Technique	Attack Target									
	Code Pointer					Data				
	Return Address	Frame Pointer	Function Pointer		Dynamic Linker Tables	Pointer		Non-Pointer		
			Stack	Static/Heap		Stack	Static/Heap	Stack	Static	Heap
StackGuard [11]	$\surd^{*}$									
Libverify [6], RAD [9]	$\surd^{\dagger\dagger}$									
Etoh and Yoda [16]	$\surd^{*}$	$\surd^{*}$	$\surd^{*}$			$\surd^{*}$		$\surd^{*}$		
PointGuard [13]	$\surd^{\ddagger}$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$			
Address Obfuscation	$\surd^{\ddagger}$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd^{\dagger}$	$\surd^{**}$	$\surd^{\dagger}$

$*$ Only protected from buffer-overflow attacks; no protection from other attacks.
† Limited protection provided.
$**$ Possible in principle but not currently implemented.
†† Susceptible to attacks that simultaneously corrupt return address and another location (second copy of return address)
‡ Some susceptibility to attacks that corrupt return address and another stack-resident pointer.

Figure 2: Targets of memory error exploits, and effectiveness of defenses against them.

tacks which exploit memory programming errors, including stack smashing, heap-overflow, integer overflow, and typical format-string attacks.

- It can be easily applied to existing legacy code without modifying the source code, or the underlying operating system. Moreover, it can be applied selectively to protect security-critical applications without needing to change the rest of the system.
- The transformation is fast and introduces only a low runtime overhead.

Applicability to legacy code without source-code or operating system changes provides an easy migration path for existing systems to adopt address obfuscation. Such a solution can also be ported more easily to proprietary operating systems. Finally, the approach can be easily combined with existing techniques, such as Stackguard and Formatguard, to provide additional security.

1.1 Overview of Address Obfuscation and How it Works.

We start with the observation that the goal of an attacker is to cause the target program to execute attack-effecting code. This code itself may be provided by the attacker (*injected code*), or it may already be a part of the program (*existing code*). A direct way to force execution of such code is through a change to the control flow of the program. This requires the attacker to change a code pointer stored somewhere in memory, so that it points to the code of their choice. In such a case, when the corrupted code pointer is used as the target of a jump or call instruction, the program ends up executing the code chosen by the attacker. Some natural choices for such code pointers include the return address (stored on the stack), function pointers (stored on the stack, static area or the heap), the global offset table (GOT) that is used in the context of dynamic linking, and buffers storing `longjmp` data. An indirect way to force execution of attack-effecting code is to change security-critical data that is used by the program in its normal course of execution. Examples of such data include arguments to a `chmod` or `execve` system call, variables holding security critical data such as a flag indicating whether a user has successfully authenticated herself, etc.

There are essentially two means by which an attacker can exploit a memory error: by overwriting a pointer value, or by overwriting non-pointer data. Since code sections cannot be overwritten in most modern operating systems, there are three possible combinations of goals and means: corrupting a code-pointer, corrupting a data-pointer, or corrupting non-pointer data. Of these, the two pointer-corrupting attacks involve overwriting a pointer with the address of data or code chosen by the attacker. These two kinds of attacks require the attacker to know the absolute address of such data or code, and hence we call them *absolute address-dependent* attacks. The third kind of attack is called *relative address-dependent*, because it does not overwrite pointers, and requires only relative address information — in particular, an attacker needs to know the relative distance between a buffer (which is overrun) and the location of the data item to be corrupted. Figure 2 shows these three classes of attacks, further subdivided based on the pointer or data value that is targeted by an attack. It shows which of today's protection schemes (including ours) protect against them. As it shows, Stackguard, Libverify and RAD protect against buffer overrun attacks that overwrite the return address. PointGuard [13] is an approach that encrypts stored pointer values (by xor-ing them with a random number). It can be thought of as obfuscating pointer values as opposed to the addresses pointed by them. The benefit of their approach is that the probability of an attack making a successful guess is smaller than with ad-

dress obfuscation. A drawback is that it does not provide protection against attacks that modify non-pointer values, e.g., attacks that modify critical data, or integer subscripts. A concrete example of such an attack is the recent integer overflow exploit [20], which is protected by address obfuscation. The PaX project's ASLR approach provides protection against pointer-based attacks in much the same way as address obfuscation, but not against data attacks that exploit relative distances between variables. A more detailed comparison of our approach with these approaches can be found in Sections 5 and 3.

1.2 Organization of the Paper.

The rest of this paper is organized as follows. In Section 2, we describe several possible obfuscating transformations, and describe our implementation approach. Section 3 discusses the effectiveness of our approach against different attacks, and analyzes the probability of mounting successful attacks. Runtime overheads introduced by our approach are discussed in Section 4, followed by a discussion of related work in Section 5. Finally, Section 6 provides a summary and discusses future work.

2 Address Obfuscation

2.1 Obfuscating Transformations

The objectives of address obfuscation are to (a) randomize the absolute locations of all code and data, and (b) randomize the relative distances between different data items. These objectives can be achieved using a combination of the following transformations:

I. Randomize the base addresses of memory regions.

By changing the base addresses of code and data segments by a random amount, we can alter the absolute locations of data resident in each segment. If the randomization is over a large range, say, between 1 and 100 million, the virtual addresses of code and data objects become highly unpredictable. Note that this does not increase the physical memory requirements; the only cost is that some of the virtual address space becomes unusable. The details depend on the particular segment:

1. *Randomize the base address of the stack.* This transformation has the effect of randomizing all the addresses on the stack. A classical stack-smashing attack requires the return address on the stack to be set to point to the beginning of a stack-resident buffer into which the attacker has injected his/her code. This becomes very difficult when the attacker cannot predict the address of such a buffer due to randomization of stack addresses. Stack-address randomization can be implemented by subtracting a large random value from the stack pointer at the beginning of the program execution.
2. *Randomize the base address of the heap.* This transformation randomizes the absolute locations of data in the heap, and can be performed by allocating a large block of random size from the heap. It is useful against attacks where attack code is injected into the heap in the first step, and then a subsequent buffer overflow is used to modify the return address to point to this heap address. While the locations of heap-allocated data may be harder to predict in long-running programs, many server programs begin execution in response to a client connecting to them, and in this case the heap addresses can become predictable. By randomizing the base address of the heap, we can make it difficult for such attacks to succeed.
3. *Randomize the starting address of dynamically-linked libraries.* This transformation has the effect of randomizing the location of all code and static data associated with dynamic libraries. This will prevent *existing code* attacks (also called *return-into-libc* attacks), where the attack causes a control flow transfer to a location within the library that is chosen by the attacker. It will also prevent attacks where static data is corrupted by first corrupting a pointer value. Since the attacker does not know the absolute location of the data that he/she wishes to corrupt, it becomes difficult for him/her to use this strategy.
4. *Randomize the locations of routines and static data in the executable.* This transformation has the effect of randomizing the locations of all functions in the executable, as well as the static data associated with the executable. The effect is similar to that of randomizing the starting addresses of dynamic libraries.

We note that all of the above four transformations are also implemented in the PaX ASLR system, but their implementation relies on kernel patches rather than program transformations. The following two classes of transformations are new to our system. They both have the effect of randomizing the relative distance between the locations of two routines, two variables, or between a variable and a routine. This makes it difficult to develop successful attacks that rely on adjacencies between data items or routines. In addition, it introduces additional randomization into the addresses, so that an attacker that has somehow learned the offsets of the base addresses will still have difficulty in crafting successful attacks.

II. Permute the order of variables/routines.

Attacks that exploit relative distances between objects, such as attacks that overflow past the end of a buffer to

overwrite adjacent data that is subsequently used in a security-critical operation, can be rendered difficult by a random permutation of the order in which the variables appear. Such permutation makes it difficult to predict the distance accurately enough to selectively overwrite security-critical data without corrupting other data that may be critical for continued execution of the program. Similarly, attacks that exploit relative distances between code fragments, such as partial pointer overflow attacks (see Section 3.2.3), can be rendered difficult by permuting the order of routines. There are three possible rearrangement transformations:

1. *permute the order of local variables in a stack frame*
2. *permute the order of static variables*
3. *permute the order of routines in shared libraries or the routines in the executable*

III. Introduce random gaps between objects.

For some objects, it is not possible to rearrange their relative order. For instance, local variables of the caller routine have to appear at addresses higher than that of the callee. Similarly, it is not possible to rearrange the order of malloc-allocated blocks, as these requests arrive in a specific order and have to be satisfied immediately. In such cases, the locations of objects can be randomized further by introducing random gaps between objects. There are several ways to do this:

1. *Introduce random padding into stack frames.* The primary purpose of this transformation is to randomize the distances between variables stored in different stack frames, which makes it difficult to craft attacks that exploit relative distances between stack-resident data. The size of the padding should be relatively small to avoid a significant increase in memory utilization.
2. *Introduce random padding between successive* `malloc` *allocation requests.*
3. *Introduce random padding between variables in the static area.*
4. *Introduce gaps within routines, and add jump instructions to skip over these gaps.*

Our current implementation supports all the above-mentioned transformations for randomizing the base addresses of memory regions, none of the transformations to reorder variables, and the first two of the transformations to introduce random gaps.

2.2 Implementation Issues

There are two basic issues concerning the implementation of the above-mentioned transformations. The first concerns the timing of the transformations: they may be performed at compile-time, link-time, installation-time, or load-time. Generally speaking, higher performance can be obtained by performing transformations closer to compilation time. On the other hand, by delaying transformations, we avoid making changes to system tools such as compilers and linkers, which makes it easier for the approach to be accepted and used. Moreover, performing transformations at a later stage means that the transformations can be applied to proprietary software that is distributed only in binary form.

The second implementation issue is concerned with the time when the randomization amounts are determined. Possible choices here are (a) transformation time, (b) beginning of program execution, and (c) continuously changing during execution. Clearly, choice (c) increases the difficulty of attacks, and is hence preferred from the point of security. Choices (a) or (b) may be necessitated due to performance or application binary interface compatibility considerations. For instance, it is not practical to remap code at different memory locations during program execution, so we cannot do any better than (b) for this case. In a similar manner, adequate performance is difficult to obtain if the relative locations of variables with respect to some base (such as the frame pointer for local variables) is not encoded statically in the program code. Thus, we cannot do any better than choice (a) in this case. However, choice (a) poses some special problems: it allows an attacker to gradually narrow down the possibilities with every attack attempt, since the same code with the same randomizations will be executed after a crash. To overcome this problem, our approach is to periodically re-transform the code. Such retransformation may take place in the background after each execution, or it may take place after the same code is executed several times. With either approach, there still remains one problem: a local attacker with access to such binaries can extract the random values from the binary, and use them to craft a successful attack. This can be mitigated by making such executables unreadable to ordinary users. However, Linux currently makes the memory maps of all processes to be readable (through the special file `/proc/pid/maps`), which means a local user can easily learn the beginning of each memory segment, which makes it much easier to defeat address obfuscation. In particular, the attacker can easily figure out the locations of the code segments, which makes it possible to craft existing code attacks. This is a limitation of our current implementation.

Our approach is to delay the transformation to the latest possible stage where adequate performance is obtainable. In our current implementation, the transformation is performed on object files (i.e., at link-time) and executables. For ease of implementation, we have

fixed many randomizations at transformation time, such as the gaps introduced within the stack frame for any given function, the locations where libraries are loaded, etc. This means that programs have to be periodically (or frequently) re-obfuscated, which may be undesirable as the obfuscation interacts with other security procedures such as integrity-checking of executables. We therefore plan to move towards options (b) and (c) in the future.

Next, we describe our approach for implementing most of the above-mentioned transformations. Our implementation targets the Intel x86 architectures running ELF-format [30] executables on the Linux operating system.

2.3 Implementation Approach

Our implementation transforms programs at the binary level, inserting additional code with the LEEL binary-editing tool [40]. The main complication is that on most architectures, safe rewriting of machine code is not always possible. This is due to the fact that data may be intermixed with code, and there may be indirect jumps and calls. These two factors make it difficult to extract a complete control-flow graph, which is necessary in order to make sure that all code is rewritten as needed, without accidentally modifying any data. Most of our transformations, such as stack base randomization are simple, and need to be performed in just one routine, and hence are not impacted by the difficulty of extracting an accurate control-flow graph. However, stack-frame padding requires a rewrite of all the routines in the program and libraries, which becomes a challenge when some routines cannot be accurately analyzed. We take a conservative approach to overcome this problem, rewriting only those routines that can be completely analyzed. Further details can be found in Section 2.3.4.

2.3.1 Stack base address randomization

The base address of the stack is randomized by extra code which is added to the text segment of the program. The code is spliced into the execution sequence by inserting a jump instruction at the beginning of the main routine. The new code generates a random number between 1 and 10^8, and decrements the stack pointer by this amount. In addition, the memory region corresponding to this "gap" is write-protected using the `mprotect` system call. The write-protection ensures that any buffer overflow attacks that overflow beyond the base of the stack into the read-only region will cause the victim program to crash.

2.3.2 DLL base address randomization

In the ELF binary format, the program header table (PHT) of an executable or a shared library consists of a set of structures which hold information about various segments of a program. Loadable segments are mapped to virtual memory using the addresses stored in the `p_vaddr` fields of the structures (for more details, see [30]). Since executable files typically use (non-relocatable) absolute code, the loadable segments must reside at addresses specified by `p_vaddr` in order to ensure correct execution.

On the other hand, shared object segments contain position-independent code (PIC), which allows them to be mapped to almost any virtual address. However, in our experience, the dynamic linker almost always chooses to map them starting at `p_vaddr`, e.g., this is the case with `libc.so.6` (the Standard C library) on Red Hat Linux distributions. The lowest loadable segment address specified is `0x42000000`. Executables start at virtual address `0x08048000`, which leaves a large amount of space (around 927MB) between the executable code and the space where shared libraries are mapped. Typically, every process which uses the dynamically-linked version of `libc.so.6` will have it mapped to the same base address (`0x42000000`), which makes the entry points of the `libc.so.6` library functions predictable. For example, if we want to know the virtual address where function `system()` is going to be mapped, we can run the following command:

```
$ nm /lib/i686/libc.so.6 | grep system
42049e54 T __libc_system
2105930 T svcerr_systemerr
42049e54 W system
```

The third line of the output shows the virtual address where `system` is mapped.

In order to prevent existing code attacks which jump to library code instead of injected code, the base address of the libraries should be randomized. There are two basic options for doing this, depending on when the randomization occurs. The options are to do the randomization (1) once per process invocation, or (2) statically. The trade-offs involved are as follows:

1. *Dynamically randomize library addresses using* `mmap`. The dynamic linker uses the `mmap` system call to map shared libraries into memory. The dynamic linker can be instrumented to instead call a wrapper function to `mmap`, which first randomizes the load address and then calls the original `mmap`. The advantage of this method is that in every program execution, shared libraries will be mapped to different memory addresses.
2. *Statically randomize library addresses at link-time.* This is done by dynamically linking the executable with a "dummy" shared library. The dummy library need not be large enough to fill the virtual address

space between the segments of the executable and standard libraries. It can simply introduce a very large random gap (sufficient to offset the base addresses of the standard libraries) between the load-addresses of its `text` and `data` segments. Since shared libraries use relative addressing, the segments are mapped along with the gap.

On Linux systems, the link-time gap can be created by using the `ld` options `-Tbss`, `-Tdata` and `-Ttext`. For example, consider a dummy library which is linked by the following command:

```
$ ld -o libdummy.so -shared
        dummy.o  -Tdata 0x20000000
```

This causes the load address of the text segment of `libdummy.so` to be `0x00000000` and the load address of data segment to be `0x20000000`, creating a gap of size `0x20000000`. Assuming the text segment is mapped at address `0x40014000` (Note: addresses from `40000000` to `40014000` are used by the dynamic linker itself: `/lib/ld-2.2.5.so`), the data segment will be mapped at address `0x60014000`, thereby offsetting the base address of `/lib/i686/libc.so.6`.

The second approach does not provide the advantage of having a freshly randomized base address for each invocation of the program, but does have the benefit that it requires no changes to the loader or rest of the system. We have used this approach in our implementation. With this approach, changing the starting address to a different (random) location requires the library to be re-obfuscated (to change its preferred starting address).

2.3.3 Text/data segment randomization

Relocating a program's text and data segments is desirable in order to prevent attacks which modify a static variable or jump to existing program code. The easiest way to implement this randomization is to convert the program into a shared library containing position-independent code, which, when using `gcc`, requires compiling with the flag `-fPIC`. The final executable is created by introducing a new `main` function which loads the shared library generated from the original program (using `dlopen`) and invokes the original `main`. This allows random relocation of the original program's text and data segments. However, position-independent code is less efficient than its absolute address-dependent counterpart, introducing a modest amount of extra overhead.

An alternative approach is to relocate the program's code and data at link-time. In this case, the code need not be position-independent, so no performance overhead is incurred, Link-time relocation of the starting address of the executable can be accomplished by simple modifications to the scripts used by the linker.

Our implementation supports both of these approaches. Section 4 presents the performance overheads we have observed with each approach.

2.3.4 Random stack frame padding

Introducing padding within stack frames requires that extra storage be pushed onto the stack during the initialization phase of each subroutine. There are two basic implementation issues that arise.

The first issue is the randomization of the padding size, which could be static or dynamic. Static randomization introduces practically no runtime overhead. Dynamic randomization requires the generation of a random number at regular intervals. Additionally, the amount of extra code required for each function preamble is significant. Moreover, if the randomization changes the distance between the base of the stack frame and any local variable (from one invocation of a function to the next) then significant changes to the code for accessing local variables are required, imposing even more overheads. For these reasons, we have currently chosen to statically randomize the padding, with a different random value used for each routine.

The second issue concerns the placement of the padding. As shown in Figure 3, there are two basic choices: (1) between the base pointer and local variables, or (2) before parameters to the function:

1. *Between the base pointer and local variables.*
   This requires transformation of the callee to modify the instruction which creates the space for the local variables on the stack. Local variables are accessed using instructions containing fixed constants corresponding to their offset from the base pointer. Given that the padding is determined statically, the transformation simply needs to change the constants in these instructions. The main benefit of this approach is that it introduces a random gap between local variables of a function and other security-critical data on the stack, such as the frame pointer and return address, and hence makes typical stack-smashing attacks difficult.
2. *Before parameters to the function.*
   This is done by transforming the caller. First, the set of argument-copying instructions is located (usually `PUSH` instructions). Next, padding code is inserted just before these instructions. The primary advantage of this approach is that the amount of padding can change dynamically. Disadvantages of the approach are (a) in the presence of optimization, the argument-pushing instructions may not be contigu-

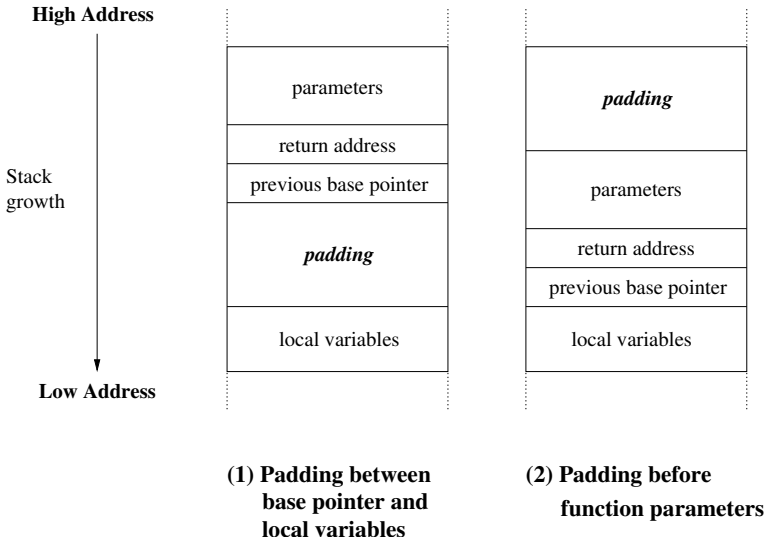

Figure 3: Potential locations of padding inserted between stack frames.

ous, which makes it difficult to determine where the padding is to be introduced, and (b) it does not make stack-smashing attacks any harder since the distance between the local variables and return address is left unchanged.

We have implemented the first option. As mentioned earlier, extraction of accurate control-flow graphs can be challenging for some routines. To ensure that our transformation does not lead to an erroneous program, the following precautions are taken:

- Transformation is applied to only those routines for which accurate control-flow graphs can be extracted. The amount of padding is randomly chosen, and varies from 0 to 256, depending on the amount of storage consumed by local variables, and the type of instructions used within the function to access local variables (byte- or word-offset). From our experience on instrumentation of different binaries, we have found that around $95 - 99\%$ of the routines are completely analyzable.
- Only functions which have suitable behavior are instrumented. In particular, the function must have at least one local variable and manipulate the stack in a standard fashion in order to be instrumented. Moreover, the routines should be free of non-standard operations that reference memory using relative addressing with respect to the frame pointer.
- Only *in place* modification of the code is performed. By *in place*, we mean that the memory layout of the routines is not changed. This is done in order to avoid having to relocate the targets of any indirect calls or jumps.

These precautions have limited our approach to instrument only 65% to 80% of the routines. We expect that this figure can be improved to 90+% if we allow modifications that are not in-place, and by using more sophisticated analysis of the routines.

2.3.5 Heap randomization

The base address of the heap can be randomized using a technique similar to the stack base address randomization. Instead of changing the stack pointer, code is added to allocate a randomly-sized large chunk of memory, thereby making heap addresses unpredictable. In order to randomize the relative distances between heap data, a wrapper function is used to intercept calls to `malloc`, and randomly increase the sizes of dynamic memory allocation requests by 0 to 25%. On some OSes, including Linux, the heap follows the data segment of the executable. In this case, randomly relocating the executable causes the heap to also be randomly relocated.

3 Effectiveness

Address obfuscation is not a foolproof defense against all memory error exploits, but is instead a probabilistic technique which increases the amount of work required before an attack (or sequence of attacks) succeeds. Hence, it is critical to have an estimate of the increase in attacker work load. In this section, we first analyze the effectiveness of address obfuscation against previously reported attacks and attack variations ("classic" attacks). Then we discuss attacks that can be specifically crafted to exploit weaknesses of address obfuscation.

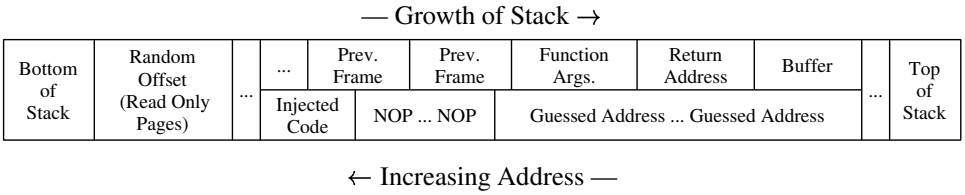

Figure 4: Format of an attack which uses a large buffer overflow to increase the odds of success.

3.1 Classic Attacks

Address obfuscation provides good protection against the majority of the "classic" attacks. Most of these attacks involve overwriting of a single pointer or datum without any ability to read the memory contents before attacking. Against address obfuscation, an attacker is forced to make guesses about the address of one or more program values in order to succeed.

3.1.1 Stack Smashing Attacks

A classic stack-smashing attack is absolute address-dependent, since the absolute address of the injected code must be placed in the return address stored in the stack frame. Let N be the size of the virtual address space available for the initial random stack offset, and assume that the stack offset is chosen randomly from $\{0 \ldots N-1\}$ (with a uniform distribution). Furthermore, we don't wish to allow an offset of zero, and Linux requires that the stack pointer be a 32-bit word-aligned address, which reduces the set of possible offsets to $\{4, 8, \ldots N\}$. (In this analysis, we assume that the one-time offset N is much larger than the effect of stack-frame padding, and hence ignore the latter. The purpose of stack-frame padding is to introduce significant additional randomization into the addresses so that attacks become difficult even if an attacker has somehow learned the value of N.)

Assuming the attacker knows the value of N, the attacker can guess an address randomly and have a $\frac{4}{N}$ chance of success. Moreover, if the guess happens to be wrong, then the program will likely crash, and will have to be restarted. At this time, a new random value for stack offset will be generated, which means that each failure does not provide any information to the attacker. Thus, the probability of a successful attack after k attempts is given by $1-(1-\frac{4}{N})^k$. From this, it can be shown that the probability of success approaches 0.5 after about $\frac{N}{8}$ attempts.

The attacker can improve the odds of success by increasing the size of the attack data. This can be done by writing to the buffer a block containing copies of a guessed address G (enough copies to be relatively sure that the return address is overwritten — in our implementation, of the order of 256 copies), followed by a block of K NOPs, and then the attack code. As long as G falls somewhere in the block of NOPs (or directly equals the first instruction of the inject code), the attack will succeed. This is illustrated in Figure 4, which shows the overlap between the stack values (along the top), and the attack data (along the bottom). When the current function returns, execution will jump to the guessed address G, which the attacker hopes will be within the range of the NOPs or the first instruction of the injected code.

The insertion of K NOPs increases the odds of success by a factor of K to $\frac{4 \cdot K}{N}$, reducing the average number of attempts for a reasonable chance of success to roughly $\frac{N}{8 \cdot K}$. Fortunately, K is limited in size because the attacker must avoid writing to the read-only stack padding. If the overflow runs into the read-only region, a segmentation fault will occur, preventing the attack from succeeding. This restricts the value of K to be much smaller than N. C programs tend not to use too much stack space; in the example programs of Figure 5, the amount of average stack storage allocated ranged from 1 to 4 kilobytes. For such programs, the maximum ratio of N to K will be $2.5 \cdot 10^4$, and the odds of a single attack succeeding will be $\frac{4}{2.5 \cdot 10^4}$, resulting in about 3000 attempts, or 12 megabytes of data transmitted, for a reasonable (≈ 0.5) probability of success. While this may seem like a small number, note that:

- every failure will cause a branch to a random address, which is highly likely to cause the target program to crash, so an attacker is not simply free to keep trying different addresses until an attack attempt succeeds. Instead, the repeated crashing of the program is likely to raise suspicion of the intruder's presence.
- the total amount of data that needs to be sent by the attacker is obtained by multiplying the size of attack data by the number of attack attempts. This number will be of the order of $\frac{N}{8}$, and is largely independent of the size of data used in each attack attempt.

3.1.2 Existing code attacks

Existing code attacks, also called *return-into-libc* attacks, typically involve overwriting the return address

on the stack with the address of existing code, typically a function in the standard C-library, such as `execve`. The arguments to this function will be taken from the stack, which has been overwritten by the same buffer overflow to contain the data chosen by attacker. In order for such an attack to succeed, the attacker needs to guess the location of the vulnerable function. With a randomization of the order of 100MB, and given the constraint that the base addresses of libraries and the executable must start at a multiple of page size (4KB), the probability of success is of the order of $4 \cdot 10^{-5}$.

Attacks that corrupt other stack-resident function pointers are all similar to an existing code attack, and the probability of a successful attack remains the same as with existing code attacks.

3.1.3 Format-String Attacks

A format-string vulnerability [33] occurs whenever a program contains a call to the `printf` family of functions with a first parameter (format string) that is provided by an attacker. Since the format string provides a great deal of control over the behavior of `printf` function, the ability of an attacker to provide a format string can be likened to the ability to execute attacker-chosen code. For this reason, most techniques developed to deal with buffer overflows are not effective against format string attacks.

The common form of this attack uses the somewhat obscure `%n` format parameter, which takes a pointer to an integer as an argument, and writes the number of bytes printed so far to the location given by the argument. The number of bytes printed can be easily controlled by printing an integer with a large amount of padding, e.g., `%432d`. The `printf` function assumes that the address to write into is provided as an argument, i.e., it is to be taken from the stack. If the attacker-provided format string is stored on the stack, and if printf can be tricked into extracting arguments from this portion of the stack, then it is possible for an attacker to overwrite an arbitrary, attacker-specified location in memory with attacker-specified data. Such an attack can be used to change return values without trampling over canary values used by StackGuard and other approaches.

The format-string attack described above is an absolute-address dependent attack. It requires the attacker to know the absolute location where the return address is stored on the stack, and the absolute location where the attack code is present. This means that the probability of a successful attack using this approach cannot be any larger than that for stack-smashing attacks.

Certain kinds of format-string vulnerabilities can be exploited to read stack contents. In particular, if the vulnerable `printf` (or variant) call is one that sends its output to the attacker, then the attacker can potentially learn the randomizations used in the program, and use this knowledge to craft a successful attack. (See Section 3.2.1 for details.)

3.1.4 Data Modification Attacks

Attacks which target non-pointer data values are one of the most difficult to defend against. For instance, a string which contains a shell command may be stored adjacently to the end of a buffer with an overflow vulnerability. In this case, an attacker can overflow the buffer with ASCII text containing a different command to be executed. The success of the attack depends only upon the relative distance between the buffer and the command string. Furthermore, even if the relative distance is randomized, the attacker can use blank characters as padding to increase the odds of success. If the attacker pads the injected string with more blanks than the maximum increase in distance between the buffer and the shell string, then the odds of success are high, especially when the data is located in the static area. If it is located on the stack, then the introduction of blanks (or other padding characters) may corrupt critical data on the stack, which may cause the program to crash. For this reason, such padding may not be very successful for stack-resident data.

Our current implementation provides limited protection against this attack, in the case where the data resides on the stack or heap. In the case of heap, if the overflow attack overwrites critical data within the same malloc-ed block as the target of the copy operation, then randomization does not help. Otherwise malloc randomization is effective, with the effectiveness increasing proportionately with the number of malloc blocks that are overwritten by the attack. Similarly, if the buffer and vulnerable data appear on the same stack frame, our current implementation does not provide any help. However, if they reside in different stack frames, then some level of protection is available, depending on the distance between the buffer and the vulnerable data.

The scope of protection can be expanded using the technique presented in [16], where all of the sensitive data (such as function and data pointers) can be located at addresses below the starting address of any buffer. Since the overflows can only move upward in memory, they can never reach from the buffer to a sensitive data location without crossing over into previous stack frames, in which case the return address will be corrupted.

Our current implementation provides no protection against relative address-dependent overflows that corrupt data in the static area. A fuller implementation of address obfuscation, which includes reordering of static variables as well as padding between them, will indeed

provide a good degree of protection against data modification attacks in the static area.

3.1.5 Heap Overflow and Double-Free Attacks

Due to the lack of adequate checking done by `malloc` on the validity of blocks being freed, code which frees the same block twice corrupts the list of free blocks maintained by `malloc`. This corruption can be exploited to overwrite an arbitrary word of memory with an arbitrary value [2]. A heap overflow attack achieves the same effect through a buffer overflow that also corrupts the data structures maintained by `malloc` [23].

Both of these are absolute address-dependent attacks, and the protection provided by address obfuscation is quite good, as the address of a single word is randomized over $\frac{10^8}{4}$ possible values.

3.1.6 Integer Overflow Attacks

Integer overflow attacks exploit an integer overflow to bypass runtime checks in a program. Since an integer has a fixed size, an overflow during a computation causes it to change its value in an undefined manner (typically, the value "wraps around" from a large positive value to a small negative one, or vice-versa). Due to the wrap-around, boolean conditions which test the values of integers resulting from arithmetic overflow are often incorrectly evaluated. For example, if i is sufficiently large, the expression $i + 5$ can overflow, resulting in a negative value, and causing the condition $i + 5 > limit$ to evaluate to false, when it should be true. This effectively disables the bounds checking, allowing an overflow attack to be performed in spite of the bounds checking.

The level of protection provided by address obfuscation from these kinds of attack is the same as for normal buffer overflow attacks. In particular, if the target corrupted by an attack is a pointer, then the probability of a successful attack is low. This was the case with the recent Snort integer overflow vulnerability. If the attack targets security critical data, then the protection is similar to that for relative address attacks. In particular, a good degree of protection is available for heap-resident data, while the level of protection for stack resident data is some what lesser. As an example, the sshd integer overflow attack involved overwriting a critical piece of string data with a null character, which was interpreted by the sshd server to mean that no password was required for a user to log in. Address obfuscation provides a good degree of protection against such an attack, while some of the related approaches such as PointGuard can be defeated by this attack.

3.2 Specifically Crafted Attacks

We have identified three specific attacks which can be used to attempt to defeat address obfuscation when the victim program contains the "right" vulnerability. These occur when (1) a program has a bug which allows an attacker to read the memory contents, or (2) an overflow exists that can be used to modify two pointer values (a buffer pointer and a function pointer), or (3) an overflow can be used to overwrite just the lower part of a pointer. In the case of (1), the attacker can craft an attack that succeeds deterministically. In the case of (2) and (3), the probability of success is significantly higher than the classic attacks, but far from deterministic.

We note all of the attacks discussed in this section require vulnerabilities that are very uncommon. Moreover, although our current implementation is vulnerable to these attacks, a full implementation of address obfuscation, employing all of the transformations described in Section 2.1, and using dynamically changing random values, will be much less vulnerable.

3.2.1 Read/Write Attacks

If a program contains a bug which allows an attacker to print the values stored in arbitrary memory locations, then most of the existing security schemes can be compromised if there is a vulnerability somewhere in the program. In the case of address obfuscation, the attacker can compare pointer values stored in the program against a local, non-obfuscated copy, and possibly decipher the obfuscation mapping. A specific instance of this occurs when an attacker can control the format-string passed to a `printf`, provided the vulnerable print statement sends its output to the attacker [29]. Given such a vulnerability, an attacker can send a format string that would cause the stack contents to be printed. From the output, the attacker can guess with a high probability (or with certainty, if no stack frame padding is used) the locations holding saved frame pointer and return address. By comparing these values with those that can be observed on their local version of the vulnerable program that has not been obfuscated, the attacker can identify the obfuscation mapping. Armed with this mapping, the attacker can develop an attack that will succeed with a high probability. This time, the attacker will use the standard format-string attack that uses the `n%` directive.

We point out that changing just the base addresses of different memory regions, as done with PaX ASLR, does not help with this attack. Most other techniques, such as PointGuard and StackGuard are also vulnerable to this attack. In the case of PointGuard, the obfuscated stack can be compared to a non obfuscated process, and the xor mask value can be inferred. In the case of StackGuard, the stack can be examined to determine the ca-

nary value, and then stack smashing can be used.

Address obfuscation, as implemented now, seems to provide some additional protection over ASLR: it is no longer possible to deterministically identify the location of frame pointer or return address. But this added difficulty does not translate into additional protection: the format-string based read attack does not cause the program to crash, so the attacker can perform multiple attacks to read the stack multiple times until he/she can determine the frame pointer with certainty. However, if the stack-frame padding is varied continuously at runtime, then address obfuscation will provide significant degree of protection. In this case, the location of the buffer, the saved frame pointer, as well as the return address, will change between the time the attacker read the contents of the stack and the time he/she tries to modify the return address. This will significantly decrease the chances of a successful attack. Probability of a successful existing-code attack can also be decreased significantly by using the more general form of address obfuscation of code, which involves reordering routines, etc.

3.2.2 Double Pointer Attacks

A program which contains a both a (preferably stack-allocated) pointer to a buffer and a buffer overflow vulnerability can be exploited to work around obfuscation. For example, consider the following code fragment, which is similar to one suggested for defeating StackGuard [7]:

```
void
f(char *user_input1, char *user_input2) {
  char *buf1 = malloc(100);
  char buf2[100];
  strcpy(buf2, user_input1);
  strncpy(buf1, user_input2, 100);
  ...
```

The steps required to exploit this code are as follows. First, the attacker can guess an address G likely to be valid (somewhere in the heap is a good choice). Second, the first `strcpy` to `buf2` can be overflowed to overwrite the the top stack locations with G, setting both `buf1` and the saved return address to equal G. Once this is done, the `strcpy` to `buf1` will copy `user_input2` to G. `user_input2` should contain the injected code. When the function returns, it will jump to address G, which is the start of the code injected via `user_input2`.

The probability of success with this attack is proportional to the probability of guessing a valid address G in memory. This probability is small for programs that use small amounts of memory as compared to the amount of randomization. For instance, if the program uses a megabyte of memory, then the probability success (with a 100MB padding) is one in a hundred. The same line of reasoning holds with PointGuard: the attacker can overwrite `buf1` and the return address with G, but these values will be interpreted as $G \ xor \ M$ where M is the xor mask used by PointGuard to encrypt pointers. This means that the probability of success is proportional to that of guessing a G such that $G \ xor \ M$ corresponds to a writable portion of the memory. This probability is given by (size of data memory used by program)/(size of address space), a quantity that is smaller than the corresponding number for address obfuscation.

3.2.3 Partial Overwrite Attacks

A *partial overwrite attack* is an attack which overwrites only part of a targeted value. For example, under the x86 architecture, an overflow could overwrite just the least significant byte of the return address. (This is hard to achieve if the buffer overflow was the result of an unchecked `strcpy` or similar function, since the terminating null character would clobber the rest of the return address. Thus, we need a buffer overflow that does not involve strings.) Since the only transformation made to code addresses is that of changing the base address, and since the quantity of change is constrained to be a multiple of the page size (4096 bytes on Linux), the location pointed by the return address is predictable when we we change its last 8 bits.

If exploitable code (i.e., code that can be used as a target in the case of existing code attacks) can be found within 256 bytes of the return address of a function with buffer-overflow vulnerability, then this attack will work against address obfuscation. However, it is very unlikely that such exploitable code can be found, so the attack suggested in [3] is more elaborate. Specifically, the attack involves the use of a call to the `printf` function in the caller code that precedes the call to the function with buffer overflow vulnerability. The attack then modifies the return address so that a return goes to the instruction that calls `printf`. The argument of the vulnerable function, which was attacker-provided, now becomes the argument to `printf`. At this point, the attacker can print the contents of the stack and then proceed as with the case where a format string bug allowed the attacker to read the stack contents.

Note that the stack-frame padding significantly increases the difficulty of carrying out this attack. In particular, there is a significant level of uncertainty (of the order of 128 bytes) in the distance between the vulnerable buffer and the return address, which the attacker can overcome only through guessing. If additional code address obfuscation transformations are used, (for instance, reordering of routines or introducing gaps within routines) then the attack becomes even harder.

Program	Combination (1)		Combination (2)	
	% Overhead	Standard Deviation (% of mean)	% Overhead	Standard Deviation (% of mean)
tar	-1	3.4	0	5.2
wu-ftpd	0	1.4	2	2.1
gv	0	6.1	2	7.1
bison	1	2.0	8	2.3
groff	-1	1.1	13	0.7
gzip	-1	1.9	14	2.5
gnuplot	0	0.9	21	1.0

Figure 5: Performance overhead introduced by address obfuscation.

4 Performance

We have collected performance data on the implementation of randomization of different memory regions. The following randomizations were implemented:

- relocating the base of the stack, heap, and code regions
- introduction of random gaps within stack frames, and at the end of memory blocks requested by malloc. The stack frame gaps were determined statically for each routine, while the malloc gaps can change with each malloc request.

We studied two different approaches for randomizing the start address of the executable:

- *Combination 1:* static relocation performed at link-time.
- *Combination 2:* dynamic relocation performed at load-time.

Both approaches incorporate all of the transformations mentioned above. Note that dynamic relocation requires the executable be compiled into position-independent code, which introduces additional runtime overheads.

Figure 5 shows the performance overheads due to the two combinations of transformations. All measurements were taken on an 800 MHz, Pentium III, 384 MB RAM machine with Red Hat 7.3 Linux OS. Average execution (system + user) time was computed over 10 runs. The overheads measured were rounded off to the nearest integral percentage. (Further precision was meaningless, given the standard deviations shown in the table.)

From the table, we see that combination (1) incurs essentially no runtime overhead (note that the negative overheads are below the standard deviation and are hence not statistically significant).

Combination (2) has noticeable runtime overhead. This is because it requires position-independent code, which is less efficient, since it performs extra operations before every procedure call, and every access to static data. On the other hand, when code is already being distributed in DLL form, combination (2) provides broad protection against memory error exploits without any additional overhead.

5 Related Work

5.1 Runtime Guarding Against Stack-Smashing and Format String Attacks

These techniques transform or augment a program to protect the return address or other specific values from being overwritten. *Stackguard* [11] is a modified version of the *gcc* compiler in which the generated code places *canary values* around the return address at runtime, so that any overflow which overwrites the return address will also modify the canary value, enabling the overflow to be detected. *StackShield* [6] and *RAD* [9] are based upon a similar modification to the compiler, but keep a separate copy of the return address instead of using canary values. *Libsafe* and *Libverify* [6] are dynamically loaded libraries which provide protection for the return address without requiring recompilation. Etoh and Yoda [16] use a source-code transformation approach which uses both canary values and relocates stack-allocated arrays so that they cannot overflow into local variables. *FormatGuard* [12] transforms source code using a modified version of *cpp* (the C Preprocessor) combined with a wrapper function for the `printf` function, so that format-string attacks are detected at runtime.

While these techniques are useful for guarding against specific attacks, their drawback is that they can deal with only a small subset of the total set of memory exploits shown in Figure 2.

5.2 Runtime Bounds and Pointer Checking

These techniques prevent buffer overflows by checking each memory access operation that can potentially cause a memory error to ensure that it does not happen. Approaches used to insert the required checks have included source-to-source translation [25, 5], specially

modified compilers [36, 22], binary rewriting [19], and virtual machines/interpreters [24]. All of the above techniques currently suffer from significant drawbacks: runtime overheads that can often be over 100%, restriction to a subset of C-language, and changes to the memory model or pointer semantics. In contrast, the focus of this paper is on techniques that produce very low overheads and are fully compatible with all C-programs.

5.3 Compile-Time Analysis Techniques

Compile-time analysis techniques [18, 32, 37, 14, 26] analyze a program's source code to determine which array and pointer accesses are safe. While these approaches are a welcome component of any programmer's debugging arsenal, they generally suffer from one or more of the following shortcomings: they do not detect all memory errors, they generate many false positive warnings, and/or they do not scale to large programs. The focus of our work is the development of techniques that require no additional effort on the part of programmers, and hence can be applied to the vast base of existing software, in binary form, with no programmer effort.

Hybrid approaches perform runtime memory-error checking, but also use static analysis to minimize the number of checks. *CCured* [28] and Cyclone [21] are two recent examples of this approach. One difficulty with these approaches is that they are not 100% compatible with existing C-code. Moreover, they disable explicit freeing of memory, and rely on garbage collection.

5.4 Code Obfuscation

Code obfuscation [38, 10, 4] is a program transformation technique which attempts to convolute the low-level semantics of programs without affecting the user-observable behavior, making obfuscated programs difficult to understand, and thereby difficult to reverse-engineer. The key difference between program obfuscation and address obfuscation is that program obfuscation is oriented towards preventing most static analyses of a program, while address obfuscation has a more limited goal of making it impossible to predict the relative or absolute addresses of program code and data. Other analyses, including reverse compilation, extraction of flow graphs, etc., are generally not affected by address obfuscation.

5.5 Randomizing Code Transformations

As mentioned earlier, address obfuscation is an instance of the broader idea of introducing diversity in nonfunctional aspects of software, an idea suggested by Forrest, Somayaji, and Ackley [17]. Their implementation model was called a *randomizing compiler,* which can introduce randomness in several non-functional aspects of the compiled code without affecting the language semantics. As a proof of concept, they developed a modification to the gcc compiler to add a random amount of padding to each stack allocation request. This transformation defeats most stack-smashing attacks prevalent today, but does not work against the large overflow attacks of the sort described in Section 3.

In the past year or two, several researchers [8, 1, 39, 15] seem to have independently attempted to develop randomization as a practical approach to defeat buffer-overflow and related attacks. Work by Chew and Song [8] randomizes the base address of the stack, system call numbers, and library entry points, through a combination of a program loader modifications, kernel system call table modifications, and binary rewriting. Xu, Kalbarczyk, and Iyer developed *transparent runtime randomization* [39], in which the Linux kernel is modified to randomize the base address of stack, heap, dynamically loaded libraries, and GOT. The PaX project's *address space layout randomization* (ASLR) approach [1] randomizes the base address of each program region: heap, code, stack, data. Of these, the ASLR approach is the most advanced in terms of its implementation. As noted earlier, ASLR is vulnerable to attacks that rely on adjacency information such as the relative addresses between variables or code, and attacks that can provide information about the base addresses of different memory segments. The introduction of additional randomization in address obfuscation, in the form of random-sized gaps within stack frames and blocks allocated by `malloc`, reordering of (and random padding within) code and static variables, can address these weaknesses. Another important difference between the above works and ours is that our obfuscations are implemented using program transformations, whereas the other works are implemented using operating system modifications. For this reason, our approach can be more easily ported to different operating systems. Moreover, it can protect individual (security-critical) applications without having to make any changes to the rest of the system.

The PointGuard [13] approach complements ours in that it randomizes ("encrypts") stored pointer values, as opposed to the locations where objects are stored. The encryption is achieved by xor'ing pointer values with a random integer mask generated at the beginning of program execution. It shares many of the benefits (such as broad protection against a wide range of pointer-related attacks) and weaknesses (susceptibility to attacks that read victim process memory to identify the mask). The principal differences are that (a) PointGuard does not protect against attacks that do not involve pointer values,

e.g., attacks that modify security-critical data through a buffer overflow, and (b) probability of successful attacks is smaller with PointGuard than with address obfuscation since the range of randomization can be as large as the address space. It should also be noted that PointGuard is dependent on the availability of accurate type information. Many C-language features, such as the ability to operate on untyped buffers (e.g., `bzero` or `memcpy`), functions that take untyped parameters (e.g., `printf`), unions that store pointers and integer values in the same location, can make it difficult or impossible to get accurate type information, which means that the corresponding pointer value(s) cannot be protected.

6 Conclusion

We believe that address obfuscation has significant potential to constrain the increasing threat of widely spread buffer overflow-type of attacks. By randomly re-arranging the memory space that holds a computer program and its data during execution, the core vulnerability that buffer overflow attacks have been exploiting is addressed — namely, the predictable location of control information and critical data. Unlike many existing techniques, which deploy attack-specific mechanisms to overcome known attack scenarios, address obfuscation is a generic mechanism that has a broad range of application to many memory error-related attacks.

Since each system is obfuscated differently, even if an attacker successfully subverts one system on a network, the attack will have to essentially start over from scratch and make many attempts before a second system can be subverted. In the context of self-replicating attacks, this factor will greatly slow down the spread of worms and viruses. Thus, address obfuscation provides a simple and effective solution to combat the spread of viruses and worms which replicate by exploiting memory errors.

Our main goal for the future is to improve the quality of randomization that can be done at the binary level. In particular, we are interested in randomizing the relative distances between objects in all regions of a program, instead of just the stack and heap, as is the case with our current implementation. There are basically two avenues for this work. The first is a tool that works with existing binary files. Such a tool will be restricted in the types of obfuscations which can be applied, but will have a wide potential impact. Addressing relative-distance issues requires both inserting padding between and permuting the order of data and code, which requires the relocation of affected addresses. Performing these sorts of relocations on binaries is not always feasible due to the difficulty of distinguishing pointers from non-pointers, sizes of data objects, and code from data. We plan on developing better analysis tools, and combining this with a flexible transformation strategy that applies as many obfuscations as possible within the limits of the analysis.

The second avenue is to augment binaries with an extra section that contains the information required to safely perform relocations. This approach requires a relatively minor change to the compiler infrastructure, as the information required is similar to the information already being generated to support the linking of object modules. Given the extra information, a program can be obfuscated at link- or load-time in a more thorough manner which will change all relative and absolute addresses in every program region.

Acknowledgments

This research was supported in part by AFOSR grant F49620-01-1-0332, ONR University Research Initiative Grant N00140110967, and NSF grants CCR-0098154 and CCR-0208877.

References

[1] Pax. Published on World-Wide Web at URL http://pageexec.virtualave.net, 2001.

[2] Anonymous. Once upon a free .... *Phrack*, 11(57), August 2001.

[3] Anonymous. Bypassing pax aslr protection. *Phrack*, 11(59), July 2002.

[4] D. Aucsmith. Tamper-resistant software: An implementation. In Ross Anderson, editor, *Information hiding: first international workshop, Cambridge, U.K., May 30–June 1, 1996: proceedings*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1996. Springer-Verlag.

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, Orlando, Florida, 20–24 June 1994. *SIGPLAN Notices* 29(6), June 1994.

[6] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)*, pages 251–262, Berkeley, CA, june 2000.

[7] Bulba and Ki13r. Bypassing stackguard and stackshield. *Phrack*, 11(56), May 2000.

[8] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[9] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing*, page 409, Phoenix, Arizona, April 2001.

[10] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the 1998 International Conference on Computer Languages*, pages 28–38. IEEE Computer Society Press, 1998.

[11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[12] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[13] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, Washington, D.C., August 2003.

[14] Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. In *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 194–?? Springer Verlag, June 2001.

[15] Daniel C. DuVarney, R. Sekar, and Yow-Jian Lin. Benign software mutations: A novel approach to protect against large-scale network attacks. Center for Cybersecurity White Paper (prepared for Airforce Office of Scientific Research), October 2002.

[16] 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, June 2000.

[17] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, pages 67–72, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[18] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language and Design*, Atlanta, GA, May 1999.

[19] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference*, pages 125–138, Berkeley, CA, USA, January 1992. USENIX.

[20] Oded Horovitz. Big loop integer protection. *Phrack*, 11(60), December 2002.

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

[22] Robert W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In M. Kamkar and D. Byers, editors, *Third International Workshop on Automated Debugging*. Linkoping University Electronic Press, 1997.

[23] Michel Kaempf. Vudo malloc tricks. *Phrack*, 11(57), August 2001.

[24] Stephen Kaufer, Russell Lopez, and Sesha Pratap. Saber-C — an interpreter-based programming environment for the C language. In USENIX Association, editor, *Summer USENIX Conference Proceedings*, pages 161–171, Berkeley, CA, USA, Summer 1988. USENIX.

[25] Samuel C. Kendall. Bcc: run–time checking for c programs. In *Proceedings of the USENIX Summer Conference*, El. Cerrito, California, USA, 1983. USENIX Association.

[26] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[27] Mudge. How to write buffer overflows. Published on World-Wide Web at URL http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html, 1997.

[28] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages (POPL '02)*, pages 128–139, Portland, OR, January 2002.

[29] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58), Dec 2001.

[30] Mary Lou Nohr. *Understanding ELF Object Files and Debugging Tools*. Number ISBN: 0-13-091109-7. Prentice Hall Computer Books, 1993.

[31] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[32] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 182–195. ACM Press, 2000.

[33] scut. Exploting format string vulnerabilities. Published on World-Wide Web at URL http://www.team-teso.net/articles/formatstring, March 2001.

[34] Snort(tm) advisory: Integer overflow in stream4. April 2003. Published on World-Wide Web at URL http://www.kb.cert.org/vuls/id/JPLA-5LPR9S.

[35] Ssh crc32 attack detection code contains remote integer overflow. 2001. Published on World-Wide Web at URL http://www.kb.cert.org/vuls/id/945216.

[36] Joseph L. Steffen. Adding run-time checking to the portable c compiler. *Software-Practice and Experience*, 22:305–316, April 1992.

[37] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, San Diego, CA, 2000.

[38] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *International Conference on Dependable Systems and Networks*, Goteborg, Sweeden, July 2001.

[39] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. Technical Report UILU-ENG-03-2207, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, May 2003.

[40] Lu Xun. A linux executable editing library. Masters Thesis, 1999. available at http://www.geocities.com/fasterlu/leel.htm.