On the Effectiveness of Address-Space Randomization

Hovav Shacham
hovav@cs.stanford.edu

Matthew Page
mpage@stanford.edu

Ben Pfaff
blp@cs.stanford.edu

Eu-Jin Goh
eujin@cs.stanford.edu

Nagendra Modadugu
nagendra@cs.stanford.edu

Dan Boneh
dabo@cs.stanford.edu

Abstract

Address-space randomization is a technique used to fortify systems against buffer overflow attacks. The idea is to introduce artificial diversity by randomizing the memory location of certain system components. This mechanism is available for both Linux (via PaX ASLR) and OpenBSD. We study the effectiveness of address-space randomization and find that its utility on 32-bit architectures is limited by the number of bits available for address randomization. In particular, we demonstrate a *derandomization attack* that will convert any standard buffer-overflow exploit into an exploit that works against systems protected by address-space randomization. The resulting exploit is as effective as the original, albeit somewhat slower: on average 216 seconds to compromise Apache running on a Linux PaX ASLR system. The attack does not require running code on the stack.

We also explore various ways of strengthening address-space randomization and point out weaknesses in each. Surprisingly, increasing the frequency of re-randomizations adds at most 1 bit of security. Furthermore, compile-time randomization appears to be more effective than runtime randomization. We conclude that, on 32-bit architectures, the only benefit of PaX-like address-space randomization is a small slowdown in worm propagation speed. The cost of randomization is extra complexity in system support.

1 Introduction

Randomizing the memory-address-space layout of software has recently garnered great interest as a means of diversifying the monoculture of software [19, 18, 26, 7]. It is widely believed that randomizing the address-space layout of a software program prevents attackers from using the same exploit code effectively against all instantiations of the program containing the same flaw. The attacker must either craft a specific exploit for each instance of a randomized program or perform brute force attacks to guess the address-space layout. Brute force attacks are supposedly thwarted by constantly randomizing the address-space layout each time the program is restarted. In particular, this technique seems to hold great promise in preventing the exponential propagation of worms that scan the Internet and compromise hosts using a hard-coded attack [11, 31].

In this paper, we explore the effectiveness of address-space randomization in preventing an attacker from using the same attack code to exploit the same flaw in multiple randomized instances of a single software program. In particular, we implement a novel version of a return-to-`libc` attack on the Apache HTTP Server [3] on a machine running Linux with PaX Address Space Layout Randomization (ASLR) and Write or Execute Only (W⊕X) pages.

Traditional return-to-`libc` exploits rely on knowledge of addresses in both the stack and the (`libc`) text segments. With PaX ASLR in place, such exploits must guess the segment offsets from a search space of either 40 bits (if stack and `libc` offsets are guessed concurrently) or 25 bits (if sequentially). In contrast, our return-to-`libc` technique uses addresses placed by the target program onto the stack. Attacks using our technique need only guess the `libc` text segment offset, reducing the search space to an entirely practical 16 bits. While our specific attack uses only a single entry-point in `libc`, the exploit technique is also applicable to chained return-to-`libc` attacks.

Our implementation shows that buffer overflow attacks (as used by, e.g., the Slammer worm [11]) are as effective on code randomized by PaX ASLR as on non-randomized code. Experimentally, our attack takes on the average 216 seconds to obtain a remote shell. Brute force attacks, like our attack, can be detected in practice, but reasonable counter-measures are difficult to design. Taking vulnerable machines offline results in a denial of service attack, and leaving them online while a fix is sought allows the vulnerability to be exploited. The problem of detecting and managing a brute force attack is especially exacerbated by the speed of our attack. While PaX ASLR appears to provide a slowdown in attack propagation, work done by Staniford *et al.* [31] suggests that this slowdown may be inadequate for inhibiting worm propagation.

Although our discussion is specific to PaX ASLR, the attack is generic and applies to other address-space randomization systems such as that in OpenBSD. The attack also applies to any software program accessible locally or through a network connection. Our attack demonstrates what we call a *derandomization attack*; derandomization converts any standard buffer-overflow exploit into an exploit that works against systems protected by address-space randomization. The resulting exploit is as effective as the original, but slower. On the other hand, the slowdown is not sufficient to prevent its being used in worms or in a targeted attack.

In the second part of the paper, we explore and analyze the effectiveness of more powerful randomization techniques such as increasing the frequency of re-randomization and also finer grained randomizations. We show that subsequent re-randomizations (regardless of frequency) after the initial address-space randomization improve security against a brute force attack by at most a factor of 2. This result suggests that it would be far more beneficial to focus on increasing the entropy in the address-space layout. Furthermore, this result shows that our brute force attacks are still feasible against network servers that are restarted with different randomization upon crashing (unlike Apache). We also analyze the effectiveness of crash detectors in mitigating such attacks.

Our analysis suggests that runtime address-space randomization is far less effective on 32-bit architectures than commonly believed. Compile-time address-space randomization can be more effective than runtime randomization because the address space can be randomized at a much finer granularity at compile-time than runtime (e.g., by reordering functions within libraries). We note that buffer overflow mitigation techniques can prevent some attacks, including the one we present in this paper. However, overflow mitigation by itself without any address-space randomization also defeats many of these attacks. Thus, the security provided by overflow mitigation is largely orthogonal to address-space randomization.

We speculate that the most promising solution appears to be upgrading to a 64-bit architecture. Randomization comes at a cost: in both 32 and 64 bit architectures, randomized executables are more difficult to debug and support.

1.1 Related Work

Exploits Buffer overflow exploits started with simple stack smashing techniques where the return address of the current stack frame is overwritten to point to injected code [1]. After the easy stack smashing vulnerabilities were discovered and exploited, a flurry of new attacks emerged that exploited overflows in the heap [20], format string errors [28], integer overflows [35], and double-`free()` errors [2].

Countermeasures Several techniques were developed to counter stack smashing — StackGuard by Cowan *et al.* [14] detects stack smashing attacks by placing canary values next to the return address. StackShield by Vendicator [32] makes a second copy of the return address to check against before using it. These techniques are effective for reducing the number of exploitable buffer overflows but does not completely remove the threat. For example, Bulba and Kil3r [8] show how to bypass these buffer overflow defenses.

ProPolice by Etoh [16] extends the ideas behind StackGuard by reordering local variables and function arguments, and placing canaries in the stack. ProPolice also copies function pointers to an area preceding local variable buffers. ProPolice is packaged with the latest versions of OpenBSD. PointGuard by Cowan *et al.* [13] prevents pointer corruption by encrypting them while in memory and only decrypting values before dereferencing.

W$\oplus$X Pages and Return-to-libc The techniques described so far aim to stop attackers from seizing control of program execution. A orthogonal technique called W$\oplus$X nullifies attacks that inject and execute code in a process's address space. W$\oplus$X is based on the observation that most of the exploits so far inject malicious code into a process's address space and then circumvent program control to execute the injected code. Under W$\oplus$X, pages in the heap, stack, and other memory segments are marked either writable (W) or executable (X), but not both. StackPatch by Solar Designer [29] is a Linux kernel patch that makes the stack non-executable. The latest versions of Linux (through the PaX project [26]) and of OpenBSD contain implementations of W$\oplus$X. Our sample attack works on a system running PaX with W$\oplus$X.

With W$\oplus$X memory pages, attackers cannot inject and execute code of their own choosing. Instead, they must use existing executable code — either the program's own code or code in libraries loaded by the program. For example, an attacker can overwrite the stack above the return address of the current frame and then change the return address to point to a function he wishes to call. When the function in the current frame returns, program control flow is redirected to the attacker's chosen function and the overwritten portions of the stack are treated as arguments.

Traditionally, attackers have chosen to call functions in the standard C-language library, `libc`, which is an attractive target because it is loaded into every Unix program and encapsulates the system-call API by which programs access such kernel services as forking child processes and communicating over network sockets. This class of attacks, originally suggested by Solar Designer [30], is therefore known as "return-to-`libc`."

Implementations of W$\oplus$X on CPUs whose memory-management units lack a per-page execute bit — for example, current `x86` chips — incur a significant performance penalty.

Another defense against malicious code injection is randomized instruction sets [6, 21]. On the other hand, randomized instruction sets are ineffective against return-to-`libc` attacks for the same reasons as those given above for W$\oplus$X pages.

Address-Space Randomization Observe that a "return-to-`libc`" attack needs to know the virtual addresses of the `libc` functions to be written into a function pointer or return address. If the base address of the memory segment containing `libc` is randomized, then the success rate of such an attack significantly decreases. This idea is implemented in PaX as ASLR [27]. PaX ASLR randomizes the base address of the stack, heap, code, and `mmap()`ed segments of ELF executables and dynamic libraries at load and link time. We implemented our attack against a PaX hardened system and will give a more detailed description of PaX in Sect. 2.1.

Previous projects have employed address randomization as a security mechanism. Yarvin *et al.* [34] develop a low-overhead RPC mechanism by placing buffers and executable-but-unreadable stubs at random locations in the address space, treating the addresses of these buffers and stubs as capabilities. Their analysis shows that a 32-bit address space is insufficient to keep processes from guessing such capability addresses, but that a 64-bit address space is, assuming a time penalty is assessed on bad guesses.

Bhatkar *et al.* [7] define and discuss address obfuscation. Their implementation randomizes the base address of the stack, heap, and code segments and adds random padding to stack frame and `malloc()` function calls. They implemented a binary tool that rewrites executables and object files to randomize addresses. Randomizing addresses at link and compilation time fixes the randomizations when the system is built. This approach has the shortcoming of giving an attacker a fixed address-space layout that she can probe repeatedly to garner information. Their solution to this problem is periodically to "re-obfuscate" executables and libraries — that is, periodically relink and recompile executables and libraries. As pointed out in their paper, this solution interferes with host based intrusion detection systems based on files' integrity checksums. Our brute force attack works just as well on the published version of this system because their published implementation only randomizes the base address of libraries à la PaX.

Xu *et al.* [33] designed a runtime randomization system that does not require kernel changes, but is otherwise similar to PaX. The primary difference between their system and PaX is that their system randomizes the location of the Global Offset Table (GOT) and patches the Procedural Linkage Table (PLT) accordingly. Our attack also works against their system because: (1) their system uses 13 bits of randomness (3 bits less than PaX), and (2) our attack does not need to determine the location of the GOT.

2 Breaking PaX ASLR

We briefly review the design of PaX and Apache before describing our attack and experimental results.

2.1 PaX ASLR Design

PaX applies ASLR to ELF binaries and dynamic libraries. For the purposes of ASLR, a process's user address space consists of three areas, called the executable, mapped, and stack areas. The executable area contains the program's executable code, initialized data, and uninitialized data; the mapped area contains the heap, dynamic libraries, thread stacks, and shared memory; and the stack area is the main user stack.

ASLR randomizes these three areas separately, adding to the base address of each one an offset variable randomly chosen when the process is created. For the Intel `x86` architecture, PaX ASLR provides 16, 16, and 24 bits of randomness, respectively, in these memory areas. In particular, the mapped data offset, called `delta_mmap`, is limited to 16 bits of randomness because (1) altering bits 28 through 31 would limit the `mmap()` system call's ability to handle large memory mappings, and (2) altering bits 0 through 11 would cause memory mapped pages not to be aligned on page boundaries.

Our attack takes advantage of two characteristics of the PaX ASLR system. First, because PaX ASLR randomizes only the base addresses of the three memory areas, once any of the three delta variables is leaked, an attacker can fix the addresses of any memory location within the area controlled by the variable. In particular, we are interested in the `delta_mmap` variable that determines the randomized offset of segments allocated by `mmap()`. As noted above, `delta_mmap` only contains 16 bits of randomness. Because our return-to-`libc` technique does not need to guess any stack addresses (unlike traditional return-to-`libc` attacks), our attack only needs to brute force the small amount of entropy in `delta_mmap`. Our attack only requires a linear search of the randomized address space. That is, our exploit requires $2^{16} = 65,536$ probes at worst and 32,768 probes on the average, which is a relatively small number.

Second, in PaX each offset variable is fixed throughout a process's lifetime, including any processes that `fork()` from a parent process. Many network daemons, specifically the Apache web server, fork child processes to handle incoming connections, so that determining the layout of any one of these related processes reveals that layout for all of them. Although this behavior on `fork()` is not a prerequisite for our attack, we show in Sect. 3.2 that it halves the expected time to success.

2.2 Return-to-libc Attack

We give a high level overview of the attack before describing its implementation in greater detail and giving experimental data. We emphasize that although our discussion is specific to PaX ASLR, the attack applies to other address-space randomization systems such as that in OpenBSD.

2.2.1 Overview

We implemented our attack on the Apache web server running on Linux with PaX ASLR and W$\oplus$X pages. The current version of the Apache server (1.3.29) has no known overflows, so we replicated a buffer overflow similar to one discovered in the Oracle 9 PL/SQL Apache module [10, 22]. This Oracle hole can be exploited using a classic buffer overflow attack — an attacker injects her own code by supplying an arbitrarily long request to the web server that overflows an internal buffer. Nevertheless, this attack fails in an Apache server protected by PaX W$\oplus$X. Instead, we exploit this hole using the return-to-`libc` technique discussed in Sect. 1.1.

Our return-to-`libc` technique is non-standard. Chained return-to-`libc` attacks generally rely on prior knowledge of stack addresses. PaX randomizes 24 bits of stack base addresses (on `x86`), making these attacks infeasible. However, PaX does not randomize the stack *layout*, which allows us to locate a pointer to attacker supplied data on the stack. Moreover, a randomized layout would provide no protection against access to data in the top stack frame, and little protection against access to data in adjacent frames.

Our attack against Apache occurs in two steps. We first determine the value of `delta_mmap` using a brute force attack that pinpoints an address in `libc`. Once the `delta_mmap` value is obtained, we mount a return-to-`libc` attack to obtain a shell.

First, the attack repeatedly overflows the stack buffer exposed by the Oracle hole with guesses for the address of the `libc` function `usleep()` in an attempt to return into the `usleep()` function. An unsuccessful guess causes the Apache child process to crash, and the parent process to fork a new child in its place, with the same randomization deltas. A successful exploit causes the connection to hang for 16 seconds and gives enough information for us to deduce the value of `delta_mmap`. Upon obtaining `delta_mmap`, we now know the location of all functions in `libc`, including the `system()` function.[1] With this information, we can now mount a return-to-`libc` attack on the same buffer exposed by the Oracle hole to invoke the `system()` function.

Our attack searches for `usleep()` first only for convenience; it could instead search directly for `system()` and check periodically whether it has obtained a shell. Our attack can therefore be mounted even if `libc` entry points are independently randomized, a possibility we consider in Sect. 3.3.2.

2.2.2 Implementation

We first describe the memory hole in the Oracle 9 PL/SQL Apache module.

Oracle Buffer Overflow We create a buffer overflow in Apache similar to one found in Oracle 9 [10, 22]. Specifically, we add the following lines to the function `ap_getline()` in `http_protocol.c`:

```
char buf[64];
      ⋮
strcpy(buf,s); /* Overflow buffer */
```

Although the buffer overflow in the Oracle exploit is 1000 bytes long, we use a shorter buffer for the sake of brevity. In fact, a longer buffer works to the attacker's advantage because it gives more room to supply shell commands.

Precomputing `libc` Addresses In order to build the exploit, we must first determine the offsets of the functions `system()`, `usleep()`, and a `ret` instruction in the `libc` library. The offsets are easily obtained using the system `objdump` tool. With these offsets, once the exploit determines the address of `usleep()`, we can deduce the value of `delta_mmap` followed by the correct virtual addresses of `system()` and `ret`, with the simple sum

$$\text{address} = \texttt{0x40000000} + \text{offset} + \texttt{delta\_mmap}.$$

(Here `0x40000000` is the standard base address for memory obtained with `mmap()` under Linux.)

Exploit Step 1 As mentioned in the overview, the first step is to determine the value of `delta_mmap`. We do this by repeatedly overflowing the stack buffer exposed by the Oracle hole with guesses for `usleep()`'s address in an attempt to return into the `usleep()` function in `libc`. More specifically, the brute force attack works as follows:

1. Iterate over all possible values for `delta_mmap` starting from 0 and ending at 65535.

[1] The `system()` function executes user-supplied commands via the standard shell (usually `/bin/sh`).

top of stack (higher addresses)
⋮
`ap_getline()` arguments
saved `EIP`
saved `EBP`
64 byte buffer
⋮
bottom of stack (lower addresses)

Figure 1: Apache child process stack before probe

top of stack (higher addresses)
⋮
`0x01010101`
`0xdeadbeef`
guessed address of `usleep()`
`0xdeadbeef`
64 byte buffer, now filled with A's
⋮
bottom of stack (lower addresses)

Figure 2: Stack after one probe

2. For each value of `delta_mmap`, compute the guess for the randomized virtual address of `usleep()` from its offset.
3. Create the attack buffer (described later) and send it to the Apache web server.
4. If the connection closes immediately, continue with the next value of `delta_mmap`. If the connection hangs for 16 seconds, then the current guess for `delta_mmap` is correct.

The contents of the attack buffer sent to Apache are best described by illustrations of the Apache child process's stack before and after overflowing the buffer with the current guess for `usleep()`'s address. Figure 1 shows the Apache child process's stack before the attack is mounted and Figure 2 shows the same stack after one guess for the address of `usleep()`.

The saved return address of `ap_getline()` (saved `EIP`) is overwritten with the guessed address of the `usleep()` function in the `libc` library, the saved `EBP` pointer is overwritten with `usleep()`'s return address `0xdeadbeef`, and `0x01010101` (decimal 16,843,009) is the argument passed to `usleep()` (the sleep time in microseconds). Any shorter time interval results in null bytes being included in the attack buffer.[2] Note that the method for placing null bytes onto the stack by Nergal [24] is infeasible because stack addresses are strongly randomized. Finally, when `ap_getline()` returns, control passes to the guessed address of `usleep()`. If the value of `delta_mmap` (and hence the address of `usleep()`) is guessed correctly, Apache will hang for approximately 16 seconds and then terminate the connection. If the address of `usleep()` is guessed incorrectly, the connection terminates immediately. This difference in behavior tells us when we have guessed the correct value of `delta_mmap`.

Exploit Step 2 Once `delta_mmap` has been determined, we can compute the addresses of all other functions in `libc` with certainty. The second step of the attack uses the same Oracle buffer overflow hole to conduct a return-to-`libc` attack. The composition of the attack buffer sent to the Apache web server is the critical component of step 2. Again, the contents of the attack buffer are best described by illustrations of the Apache child process's stack before and after the step 2 attack. Figure 3 shows the Apache child process's stack before the attack and Figure 4 shows the stack immediately after the `strcpy()` call in `ap_getline()` (the attack buffer has already been injected).

[2] Null bytes act as C string terminators, causing `strcpy()` (our attack vector) to terminate before overflowing the entire buffer.

top of stack (higher addresses)
⋮
`ap_getline()` arguments
saved `EIP`
saved `EBP`
64 byte buffer
⋮
bottom of stack (lower addresses)

Figure 3: Apache child process stack before overflow

top of stack (higher addresses)
⋮
pointer into 64 byte buffer
`0xdeadbeef`
address of `system()`
address of `ret` instruction
⋮
address of `ret` instruction
`0xdeadbeef`
64 byte buffer (contains shell commands)
⋮
bottom of stack (lower addresses)

Figure 4: Stack after buffer overflow

The first 64 bytes of the attack buffer is filled with the shell command that we want `system()` to execute on a successful exploit. The shell command is followed by a series of pointers to `ret` instructions that serves as a "stack pop" sequence. Recall that the `ret` instruction pops 4 bytes from the stack into the `EIP` register, and program execution continues from the address now in `EIP`. Thus, the effect of this sequence of `ret`s is to pop a desired number of 32-bit words off the stack. Just above the pointers to `ret` instructions, the attack buffer contains the address of `system()`. The stack pop sequence "eats up" the stack until it reaches a pointer pointing into the original 64 byte buffer, which serves as the argument to the `system()` function. We find such a pointer in the stack frame of `ap_getline()`'s calling function.

After executing `strcpy()` on the exploited buffer, Apache returns into the sequence of `ret` instructions until it reaches `system()`. Apache then executes the `system()` function with the supplied commands. In our attack, the shell command is "`wget http://www.example.com/dropshell ; chmod +x dropshell ; ./dropshell ;`" where `dropshell` is a program that listens on a specified port and provides a remote shell with the user id of the Apache process. Note that any shell command can be executed.

2.2.3 Experiments

The brute force exploit was executed on a 2.4 GHz Pentium 4 machine against a PaX ASLR (for Linux kernel version 2.6.1) protected Apache server (version 1.3.29) running on a Athlon 1.8 GHz machine. The two machines were connected over a 100 Mbps network.

Each probe sent by our exploit program results in a total of approximately 200 bytes of network traffic, including Ethernet, IP, and TCP headers. Therefore, our brute force attack only sends a total of 12.8 MB of network data at worst, and 6.4 MB of network data on expectation.

After running 10 trials, we obtained the following timing measurements (in seconds) for our attack against the PaX ASLR protected Apache server:

Average	Max	Min
216	810	29

The speed of our attack is limited by the number of child processes Apache allows to run concurrently. We used the default setting of 150 in our experiment.

2.3 Information Leakage Attacks

In the presence of information leakage, attacks can be crafted that require fewer probes and are therefore more effective than our brute force attack in defeating randomized layouts. For instance, Durden [15] shows how to obtain the `delta_mmap` variable from the stack by retrieving the return address of the `main()` function using a format string vulnerability. Durden also shows how to convert a special class of buffer overflow vulnerabilities into a format string vulnerability.

Not all overflows, however, can be exploited to create a format string bug. Furthermore, for a remote exploit, the leaked information has to be conveyed back to the attacker over the network, which may be difficult when attacking a network daemon. Note that the brute force attack described in the previous section works against any buffer overflows and does not make any assumptions about the network server.

3 Improvements to Address-Space Randomization Architecture

Our attack on address-space randomization relied on several characteristics of the implementation of PaX ASLR. In particular, our attack exploited the low entropy (16 bits) of PaX ASLR on 32-bit x86 processors, and the feature that address-space layouts are randomized only at program loading and do not change during the process lifetime. This section explores the consequences of changing either of these assumptions by moving to a 64-bit architecture or making the randomization more frequent or more fine-grained.

3.1 64-Bit Architectures

In case of Linux on 32-bit `x86` machines, 16 of the 32 address bits are available for randomization. As our results show, 16 bits of address randomization can be defeated by a brute force attack in a matter of minutes. Any 64-bit machine, on the other hand, is unlikely to have fewer than 40 address bits available for randomization given that memory pages are usually between 4 kB and 4 MB in size. Online brute force attacks that need to guess at least 40 bits of randomness can be ruled out as a threat, since an attack of this magnitude is unlikely to go unnoticed. Although 64-bit machines are now beginning to be more widely deployed, 32-bit machines are likely to remain the most widely deployed machines in the short and medium term. Furthermore, applications that run in 32-bit compatibility mode on a 64-bit machine are no less vulnerable than when running on a 32-bit machine.

Some proposed 64-bit systems implement a global virtual address space: that is, all applications share a single 64-bit address space [12]. Analyzing the effectiveness of address randomization in these operating systems is beyond the scope of this paper.

3.2 Randomization Frequency

PaX ASLR randomizes a process's memory segments only at process creation. If we randomize the address space layout of a process more frequently, we might naively expect a significant increase in security. However, we will demonstrate that after the initial address space randomization, periodic re-randomizing adds no more than 1 bit of security against brute force attacks regardless of the frequency, providing little extra security. This also shows that brute force attacks are feasible even against non-forking network daemons that crash on every probe. On the other hand, frequent re-randomizations can mitigate the damage when the layout of a fixed randomized address space is leaked through other channels.

We analyze the security implications of increasing the frequency of address-space randomization by considering two brute force attack scenarios:

1. The address-space randomization is fixed during the duration of an attack. For example, this scenario applies to our brute force attack against the current implementation of PaX ASLR or in any situation where the randomized address space is fixed at compile-time.

2. The address-space randomization changes with each probe. It is pointless to re-randomize the address space more than once between any two probes. Therefore, *this scenario represents the best re-randomization frequency for a ASLR program.* This scenario applies, for example, to brute force attacks attacks against non-forking servers protected by PaX ASLR that crash on every probe; these servers are restarted each time with a different randomized address-space layout.

The brute force attacks in the two scenarios are different. In scenario 1, a brute force attack can linear search the address space through its probes before launching the exploit (exactly our attack in Sect. 2). In scenario 2, a brute force attack guesses the layout of the address space randomly, tailors the exploit to the guessed layout, and launches the exploit.

We now analyze the expected number of probe attempts for a brute force attack to succeed against a network server in both scenarios. In each case, let n be the number of bits of randomness that must be guessed to successfully mount the attack, implying that there are 2^n possibilities. Furthermore, only 1 out of these 2^n possibilities is correct. The brute force attack succeeds once it has determined the correct state.

Scenario 1 In this scenario, the server has a fixed address-space randomization throughout the attack. Since the randomization is fixed, we can compute the expected number of probes required by a brute force attack by viewing the problem as a standard sampling without replacement problem. The probability that the brute force attack succeeds only after taking exactly t probes is

$$\underbrace{\frac{2^n-1}{2^n}\cdot\frac{2^n-2}{2^n-1}\cdots\frac{2^n-t-1}{2^n-t}}_{\text{Pr[first } t-1 \text{ probes fail]}}\cdot\frac{1}{2^n-t-1}=\frac{1}{2^n},$$

where n is the number of bits of randomness in the address space. Therefore, the expected number of probes required for scenario 1 is

$$\sum_{t=1}^{2^n} t\cdot\frac{1}{2^n}=\frac{1}{2^n}\cdot\sum_{t=1}^{2^n} t=(2^n+1)/2\approx 2^{n-1}.$$

Scenario 2 In this scenario, the server's address space is re-randomized with every probe. Therefore, the expected number of probes required by a brute force attack can be computed by viewing the problem as a sampling with replacement problem. The probability that the brute force attack succeeds only after taking exactly t probes is given by the geometric random variable with $p = 1/2^n$. The expected number of probes required is $1/p = 2^n$.

Conclusions We can easily see that a brute force attack in scenario 2 requires approximately $2^n/2^{n-1} = 2$ times as many probes compared to scenario 1. Since scenario 2 represents the best possible frequency that an ASLR program can do, we conclude that increasing the frequency of address-space re-randomization is at best equivalent to increasing the entropy of the address space by only 1 bit.

The difference between a forking server and a non-forking server for the purposes of our brute force attack is that for the forking server the address-space randomization is the same for all the probes, whereas the non-forking server crashes and has a different address-space randomization on every probe. This difference is exactly that between scenarios 1 and 2. Therefore, the brute force attack is also feasible against non-forking servers if the address-space entropy is low. For example, in the case of Apache protected by PaX ASLR, we expect to perform $2^{15} = 32,768$ probes before fixing the value of `delta_mmap`, whereas if Apache were a single-process event-driven server that crashes on each probe, the expected number of probes required would double to a mere $2^{16} = 65,536$.

3.3 Randomization Granularity

PaX ASLR only randomizes the offset location of an entire shared library. Below, we discuss the feasibility of randomizing addresses at an even finer granularity. For example, in addition to randomizing segment base addresses, we could also randomize function and variable addresses within memory segments. Finer grained address randomization can potentially stymie brute force attacks by increasing the randomness in the address space. For example, if the `delta_mmap` variable in PaX ASLR contained 28 bits of randomness instead of 16 bits, then our brute force attack would become infeasible. We divide our analysis by considering address randomization at both runtime and compile-time.

3.3.1 Randomizing At Compile-Time

Beyond simple randomization of segments' base addresses, the compiler and linker can be easily modified to randomize variable and function addresses within their segments, or to introduce random padding into stack frames. Increasing the granularity of address-space randomization at compile and link time is easier than at the start of program execution because source code contains more relocation information than precompiled and prelinked program binaries.

Compile-time randomization was used by Bhatkar *et al.* [7] to implement address randomization with a modified compiler and linker. Unfortunately, their published implementation does not randomize more than the base addresses of library and executable segments and therefore gives no greater security than PaX ASLR against our derandomization attack.

On the other hand, compile-time randomization is not limited by the page granularity of the virtual memory system. By placing entry points in a random order within a library, a compiler can provide 10–12 additional bits of entropy (depending on architecture). Since shared libraries are by their nature shared, however, the location of entry points within these libraries can be discovered by any user or revealed by any compromised daemon on the same machine. Recompiling the standard libraries in a Unix distribution is a lengthy and computation-intensive process, and is unlikely to be undertaken frequently. However, some form of dynamic binary re-writing may be possible.

3.3.2 Randomizing at Runtime

Next we consider implementing finer granularity randomization such as function reordering within a shared library or executable at runtime.

Randomizing More than 16 Bits Since at most 16 bits of the available 32 bits are randomized by PaX ASLR and other ASLR systems on 32 bit architectures, we examine the possibility of increasing the number of randomized bits at runtime. On typical systems, 12 of the 32 address bits are page offset bits, which cannot be randomized at runtime without significant modification to the underlying virtual memory system or memory management hardware. From the remaining 20 bits, the PaX system randomizes only 16 bits. The top 4 bits are not randomized so as to prevent fragmentation of virtual address space. Since significant modifications to the virtual memory system are best avoided, we see that at most 20 bits can be randomized. Given our results for derandomizing 16 random bits, guessing 20 bits would take roughly only $2^4 = 16$ times longer, which is still within the range of a practical attack.

Reordering Functions At first glance, randomizing the order in which individual functions appear within a library or executable appears effective in preventing an attacker from extending knowledge of the actual address of one function in a library into knowledge of every function address in that library. Nevertheless, this technique does not make it any more difficult to guess a single function's address. Because our attack can be modified so that it only needs to find one function's address — that of the `system()` function — this technique is ineffective against such brute force attacks. On the other hand, this technique is effective against return-to-`libc` attacks that use multiple `libc` function calls, so it is worth exploring the technical issues in implementing it.

The process of compiling and linking fixes many relationships between runtime addresses. For example, in a standard shared library, the difference in the addresses of any two given functions remains constant between library loads. As a result, internal function calls within a shared library may use direct relative jumps. Such relative jumps prevent reordering of function addresses as part of dynamic linking at runtime. By modifying the compiler and linker, we can eliminate relative jumps at compile-time and defer resolution of offsets until runtime dynamic linking, which allows us to order functions arbitrarily or even load functions from one library into arbitrary, non-contiguous portions of virtual memory. The same applies to executables. Because indirect jumps through pointers are more expensive than direct relative jumps, these changes will exact a small runtime performance penalty.

A naive implementation of function address randomization runs into an additional problem: a page can only be shared among processes if it has the same content in each. Because functions are not generally page-aligned or an exact multiple of a page in length, shuffling them causes library pages to differ from one process to another. Thus, naively shuffling functions eliminates sharing, an important advantage of shared libraries, with accompanying cost in time and space. Fixing the problem is not difficult, requiring only clustering functions into page-size (or page-multiple) groups, then shuffling the groups instead of individual functions. The result yields less diversity, because fewer (and larger) units are shuffled, but should perform much better.

Finally, regardless of how well functions are randomized, code that needs to call these functions must be able to locate them quickly. In modern ELF-based Unix-like systems, shared library functions are found by consulting the Global Offset Table (GOT), an array of pointers initialized by the runtime dynamic linker. Each dynamic object normally maintains its own GOT and refers to it via relative offsets fixed at link time. Shuffling functions changes relative offsets, rendering this simple approach untenable. Thus, we need some new way to find shared library functions, either one based on the GOT or an entirely new technique.

Any acceptable replacement or fix for the GOT must satisfy several constraints. Lookups must be fast, because function calls are common and with shuffling almost every function call requires a lookup (without shuffling, only inter-library calls require lookups). Lookups must require little code because they must be inlined (otherwise we need a way to find the code to do a lookup, which is a recursive instance of our problem). The GOT replacement must not break sharing of code pages, because of the associated memory cost and cache penalties. Finally, the GOT replacement must not place data or code in fixed or easily calculated memory locations or replicate data so many times that it becomes easy to locate.

We have not found any solution that satisfies all of these constraints. If we discard the concept of a GOT entirely and use the dynamic loader to fix up addresses in objects at load time, we also prevent code page sharing. If we make multiple copies of the GOT in virtual memory, positioning one at a fixed relative offset to each code page, we substantially increase an attacker's chance of locating a copy via random probing. If we reserve a register for locating the current library's GOT, we exacerbate register scarcity on the `x86`, although we could use the frame pointer register (EBP) at the cost of making code difficult to debug. (Moreover, each library must manage its own GOT, so the value in the register must change and be restored in inter-library calls.) All other solutions we have considered are similarly problematic. Designing a linking architecture that facilitates function shuffling in shared code pages efficiently and securely is an open problem and a direction for future research.

We have seen that, by randomizing segment offsets, PaX ASLR provides approximately 16 bits of entropy against brute force attack, in either forking or non-forking servers. Designing a runtime randomization system that randomizes with page granularity but maintains fidelity to the traditional Unix dynamic-linking system is nontrivial. Also, re-randomization of a running C program is not feasible; and if it were feasible, such a technique would also not deliver additional entropy. Thus, on 32-bit systems, runtime randomization cannot provide more than 16–20 bits of entropy.

3.4 Monitoring and Catching Errors

The PaX developers suggest that ASLR be combined with "a crash detection and reaction mechanism" [27], which we call a *watcher*. An attacker who attempts to discover addresses within ASLR-protected executables will, in the process, trigger segmentation violations through his inevitably incorrect guesses. The watcher can detect these segmentation violations and take action to impede the attacker; for example, shut down the program under attack.

We do not believe that the crash watcher is a viable defense mechanism because of the limited actions the crash watcher can undertake when it discovers that a PaX-protected forking daemon is experiencing segmentation faults. Either the watcher alerts an administrator or it acts on its own. If it acts on its own, it can either shut down the daemon entirely or attempt to prevent the attacker from exploiting it.

If the watcher alerts an administrator, then it is difficult to see how the administrator can react in time. Our demonstrated attack can be completed in 216 seconds on the average, less time than would be necessary to diagnose the network traffic, read BugTraq, assess the severity of the situation, and take corrective measures. The administrator could also shut down the daemon before attempting a diagnosis, but in this case he would be acting no more intelligently than the watcher might.

If, indeed, the watcher shuts down the daemon altogether pending an administrator's attention, then it in effect acts as a force multiplier for denial of service. If Amazon.com's Apache servers are PaX-protected and watched, and a vulnerability is discovered in Apache that allows a segmentation violation to be induced, then Amazon can be taken offline persistently, with a minimum of attacker effort. Being taken offline persistently can be costly; reports in 2000 show that Amazon loses about $180,000 per hour of downtime [25].

While it may be true that Amazon would do better with disabled servers than compromised servers — that, in the end, is an economic question — it is, nevertheless, also true that it is difficult to distinguish exploitable vulnerabilities from mere (segfault-inducing) denial of service. Neither an automated watcher program nor a system administrator working under time pressure can be expected to make the correct determination.

It is worth illustrating how difficult these two cases are to distinguish, even for expert programmers. The Apache chunked-encoding vulnerability [9] was for several days believed, by the Apache developers themselves, not to be exploitable on 32-bit platforms: "Due to the nature of the overflow on 32-bit Unix platforms this will cause a segmentation violation and the child will terminate" [4]. After the release of a working exploit for 32-bit BSD platforms, the Apache developers revised their analysis: "Though we previously reported that 32-bit platforms were not remotely exploitable, it has since been proven by Gobbles that certain conditions allowing exploitation do exist" [5].

Furthermore, unless the segfault watcher shuts down the daemon permanently after a single segmentation violation, an attacker can still slip under the radar. For example, if the watcher acts after observing ten crashes in a one-minute period, the attacker can seek addresses by brute force at the rate of nine attempts per minute. The same holds if the watcher keeps a daemon shut down for several seconds after a crash. Such a watcher is, furthermore, as much a force multiplier for denial of service as one that shuts down the watched daemon after a single crash.

Finally, a watcher could attempt to prevent an attacker from exploiting a vulnerability while allowing the daemon to continue running. It might, for example, attempt to determine the network source of the offending requests and selectively firewall the source away from the daemon. But this assumes that the attacker can be effectively localized. With zombie networks numbering hundreds of thousands of compromised hosts available for use as launchpads [17, 23], attackers can design and deploy worms that attack vulnerable daemons in a coordinated fashion: no source machine needs to connect to the attacked machine more than once, so a firewalling watcher is of no value. Properly-engineered automated threats, therefore, are capable of bypassing even firewalling watchers unimpeded.

Sites that run large numbers of servers often load-balance incoming requests. In such situations clients are not always guaranteed persistent sessions with a single server, but instead get a different server assigned to each request. (Load balancing slows down our attack only by a factor of 2.) A watcher running locally on one of these servers would be unable to detect an attack, since subsequent segfault-inducing requests are likely to be routed to different servers. In order to detect such an attack, a networked watcher is required that can correlate segfault-inducing requests. Such a networked watcher would be difficult to implement and would not be much better at making watcher decisions than a host based watcher, due to the inherent difficulty of implementing a realistic watcher strategy.

In summary, the discussion above suggests that any reasonable implementation of the crash watcher suggested by the PaX documentation cannot prevent an attack such as we describe above from succeeding, except at the cost of facilitating and exacerbating denial-of-service vulnerabilities in the watched daemon.

3.5 Anti-Buffer Overflow Techniques

Overflow mitigation systems that protect the stack, such as StackGuard [14], ProPolice [16], and PointGuard [13], make it more difficult for an attacker to use a stack-based overflow to write arbitrary data onto the stack. (PointGuard also encrypts pointers.) Some vulnerabilities, including the one we exploited in Sect. 2.2, can no longer be exploited in the presence of overflow mitigation. However, overflow mitigation by itself, without address-space randomization, also defeats many of these attacks. Thus, the security provided by overflow mitigation is largely orthogonal to address-space randomization.

4 Conclusions

We showed that any buffer-overflow attack can be made to work against Apache running under PaX Address Space Layout Randomization and Write or Execute Only pages. Experimentally, our attack took, on the average, 216 seconds to obtain a remote shell.

Our exploit employed a novel return-to-`libc` technique in which it is not necessary for the attacker to guess addresses on the stack. Brute force was needed only to find a single 16-bit delta. Although our implemented exploit was specific to PaX ASLR and Apache, the attack is generic and applies to other address-space randomization systems such as that in OpenBSD. The attack also applies to any software program that accepts connections from the network. This attack is an instance of a *derandomization attack*, which converts any standard buffer-overflow exploit into an exploit that works against systems protected by address-space randomization. The resulting exploit is as effective as the original, but slower; the slowdown is not sufficient to frustrate worms or targeted attacks.

Our results suggest that, for current 32-bit architectures, (1) address-space randomization is ineffective against the possibility of generic exploit code for a single flaw; and (2) brute force attacks can be efficient, and hence effective.

In addition, we have analyzed the effectiveness of more powerful randomization techniques such as increasing the frequency and granularity of randomization. Subsequent re-randomizations (regardless of frequency) after the initial address-space randomization increases resistance to brute force attack by at most a factor of 2. We also argue that one cannot effectively prevent our attack without introducing a serious denial-of-service vulnerability.

Compile-time address-space randomization is more effective than runtime randomization because it can randomize addresses at a finer granularity, but the randomization it produces is more vulnerable to information leakage. To protect against information leakage, sensitive daemons should be placed within a `chroot` environment along with their libraries, so that user accounts and other daemons running on the same machine cannot be subverted into revealing the compile-time randomization used in the sensitive daemons. Buffer overflow mitigation techniques can protect against our attack, even in the absence of address-space randomization. Thus, the use of overflow mitigation is largely orthogonal to address-space randomization.

While compile-time and runtime randomization can be combined to yield better security, the most promising solution appears to be upgrading to a 64-bit architecture. Our attack is ineffective on 64-bit architectures. On 32-bit architectures, it is difficult to design randomized systems that resist brute force attacks. Applications that run in 32-bit compatibility mode on a 64-bit machine are no less vulnerable than when running on a 32-bit machine.

5 Acknowledgments

The authors thank Constantine Sapuntzakis for his detailed comments on the manuscript.

References

[1] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), Nov. 1996. `http://www.phrack.org/phrack/49/P49-14`.

[2] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), Aug. 2001. `http://www.phrack.org/phrack/57/p57-0x09`.

[3] Apache Software Foundation. The Apache HTTP Server project. `http://httpd.apache.org`.

[4] Apache Software Foundation. ASF bulletin 20020617, June 2002. `http://httpd.apache.org/info/security_bulletin_20020617.txt`.

[5] Apache Software Foundation. ASF bulletin 20020620, June 2002. `http://httpd.apache.org/info/security_bulletin_20020620.txt`.

[6] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. 10th ACM Conf. Comp. and Comm. Sec. — CCS 2003*, pages 281–9. ACM Press, Oct. 2003.

[7] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In V. Paxson, editor, *Proc. 12th USENIX Sec. Symp.*, pages 105–20. USENIX, Aug. 2003.

[8] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack Magazine*, 56(5), May 2000. `http://www.phrack.org/phrack/56/p56-0x05`.

[9] CERT, June 2002. `http://www.cert.org/advisories/CA-2002-17.html`.

[10] CERT. CERT advisory CA-2002-08: Multiple vulnerabilities in Oracle servers, Mar. 2002. `http://www.cert.org/advisories/CA-2002-08.html`.

[11] CERT. CERT advisory CA-2003-04: MS-SQL Server worm, Jan. 2003. `http://www.cert.org/advisories/CA-2003-04.html`.

[12] J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska. How to use a 64-bit address space. Technical Report 92-03-02, University of Washington, Department of Computer Science and Engineering, March 1992.

[13] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In V. Paxson, editor, *Proc. 12th USENIX Sec. Symp.*, pages 91–104. USENIX, Aug. 2003.

[14] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic detection and prevention of buffer-overflow attacks. In A. Rubin, editor, *Proc. 7th USENIX Sec. Symp.*, pages 63–78. USENIX, Jan. 1998.

[15] T. Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 59(9), June 2002. `http://www.phrack.org/phrack/59/p59-0x09`.

[16] H. Etoh and K. Yoda. ProPolice: Improved stack-smashing attack detection. *IPSJ SIGNotes Computer SECurity*, 014(025), Oct. 2001. `http://www.trl.ibm.com/projects/security/ssp`.

[17] FedCIRC. BotNets: Detection and mitigation, Feb. 2003. `http://www.fedcirc.gov/library/documents/botNetsv32.doc`.

[18] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In J. Mogul, editor, *Proc. 6th Work. Hot Topics in Operating Sys. — HotOS 1997*, pages 67–72. IEEE Computer Society, May 1997.

[19] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. Pfleeger, J. Quarterman, and B. Schneier. Cybersecurity: The cost of monopoly — how the dominance of Microsoft's products poses a risk to security. Technical report, Comp. and Comm. Ind. Assn., 2003.

[20] M. Kaempf. Vudo malloc tricks. *Phrack Magazine*, 57(8), Aug. 2001. `http://www.phrack.org/phrack/57/p57-0x08`.

[21] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. 10th ACM Conf. Comp. and Comm. Sec.*, pages 272–80. ACM Press, Oct. 2003.

[22] D. Litchfield. Hackproofing Oracle Application Server, Jan. 2002. `http://www.nextgenss.com/papers/hpoas.pdf`.

[23] L. McLaughlin. Bot software spreads, causes new worries. *IEEE Distributed Systems Online*, 5(6), June 2004. `http://csdl.computer.org/comp/mags/ds/2004/06/o6001.pdf`.

[24] Nergal. The advanced return-into-lib(c) exploits (PaX case study). *Phrack Magazine*, 58(4), Dec. 2001. `http://www.phrack.org/phrack/58/p58-0x04`.

[25] D. Patterson. A simple way to estimate the cost of downtime. In A. Couch, editor, *Proc. 16th Systems Administration Conf. — LISA 2002*, pages 185–8. USENIX, Nov. 2002.

[26] PaX Team. PaX. `http://pax.grsecurity.net`.

[27] PaX Team. PaX address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`.

[28] Scut/team teso. Exploiting format string vulnerabilities. `http://www.team-teso.net`, 2001.

[29] Solar Designer. StackPatch. http://www.openwall.com/linux.

[30] Solar Designer. "return-to-libc" attack. Bugtraq, Aug. 1997.

[31] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in your spare time. In D. Boneh, editor, *Proc. 11th USENIX Sec. Symp.*, pages 149–67. USENIX, Aug. 2002.

[32] Vendicator. StackShield. http://www.angelfire.com/sk/stackshield.

[33] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent runtime randomization for security. In A. Fantechi, editor, *Proc. 22nd Symp. on Reliable Distributed Systems — SRDS 2003*, pages 260–9. IEEE Computer Society, Oct. 2003.

[34] C. Yarvin, R. Bukowski, and T. Anderson. Anonymous RPC: Low-latency protection in a 64-bit address space. In *Proc. USENIX Summer 1993 Technical Conf.*, pages 175–86. USENIX, June 1993.

[35] M. Zalewski. Remote vulnerability in SSH daemon CRC32 compression attack detector, Feb. 2001. http://www.bindview.com/Support/RAZOR/Advisories/2001/adv_ssh1crc.cfm.