

# Address-Space Randomization: An Effective Implementation

Michael Cloppert (*cloppemj@gwu.edu*)  
The George Washington University

May 13, 2006

## 1 Abstract

The true protection offered by randomization of the memory address space has been widely debated, most notably by Shacham, *et al*[1]. The limited entropy afforded by memory addresses of 32-bit architectures, specifically, allows for brute-force discovery of the randomized locations of critical system objects. In this paper, it is shown that a watcher process can successfully stymie attempts to remotely discover randomized memory address offsets. In this implementation, address space randomization becomes an effective protection measure against arbitrary code execution.

## 2 Introduction

Address-space Layout Randomization, or ASLR, was promised as part of a solution that would prevent all arbitrary code execution exploits[2]. When paired with  $W\oplus X$  memory pages, the solution is indeed a strong one. However, it has been shown that a simple return-to-libc attack avoids  $W\oplus X$  protections altogether, and memory addressing limitations in 32-bit architectures lead to insufficient entropy in the layout randomization to prevent a brute-force discovery object locations in memory. It has been further suggested that a watcher process capable of identifying brute-force discovery as it takes place on a protected system would not adequately prevent the attack, due to the limited types of responses such a process could invoke [1].

The PaX and grsecurity teams[3], responsible for an early realization of ASLR, have added brute-force discovery protections to their implementation of the technology.[4] These protections, while effective at preventing brute-force discovery of memory layout, will disable a service for a predetermined period of time when the attack is identified. This essentially translates the attack from one of code execution to denial-of-service. It does not prevent the attack outright, and is not practical in high-availability or even production environments.

In this paper, it is shown that a watcher process can be trivially implemented to protect a system in a number of common attack scenarios without seriously impacting the availability of the service. After discussing some overlapping technologies and approaches in §3, the theory behind ASLR and an associated watcher process will be introduced briefly in §4. §5 discusses the implementation tested, and §6 discusses what the findings mean in a practical sense.

## 3 Related Work

Significant research has been performed in recent years on preventing arbitrary code execution. A number of approaches have been developed for randomizing the memory address space. Notably, the PaX and grsecurity teams have implemented this for Linux systems with kernel-level modifications [3, 4]. Bhatkar, et al., developed a memory address randomization technique at the process level, without kernel modifications

[5]. This technique also randomizes more elements of the address-space layout, making brute-force object location discovery more difficult, but not entirely removing it as a threat. To the author’s knowledge, no prior work relating to watcher processes enhancing address-space layout randomizations exists as of the writing of this paper.

## 4 Theory

### 4.1 Address-Space Randomization

In order to execute arbitrary code on a vulnerable process, an attacker needs to know the locations of critical objects in memory so that the functions necessary to achieve his or her goals can be called. One commonly needed memory location is that of the `libc system()` call, which will execute any system command with the privileges of the calling process. A simple attack on a vulnerable process could call `system()` with the parameter `"/bin/bash"`, directly resulting in a shell prompt with that process’s privileges. In order to call the `system()` process, the attacker must know the location of this function in memory.

Address-space randomization randomizes the locations of these memory objects in the stack and heap. This makes hard-coded memory references not portable from one system to another, and as such, an exploit written for one system cannot be executed on another[2].

When this is implemented on 32-bit systems, however, it has been shown that there is insufficient entropy in the address space to prevent discovery of needed objects in memory via brute force. In the case of PaX ASLR, the  $2^{16}$  possible locations of a function can be discovered in as little as 216 seconds[1]. The obvious options to address this shortcoming are to either increase the size of the memory space available for randomization to make a brute-force discovery prohibitively ex-

pensive, or to prevent the brute-force attack altogether. The former requires an expensive architectural change; the latter may be achievable programatically.

### 4.2 Watcher Process

A *watcher* is a process that monitors another protected and potentially vulnerable process. The goal of the watcher is to identify every memory segfault, terminated task, and invalid execution attempt that may be indicative of a brute-force discovery against the address space layout. The watcher process is therefore reliant on the system to report verbosely every time such an event takes place. Once a brute-force attack against the address space mapping is identified, the watcher process reacts by implementing blocks to prevent additional attacks from the same source.

The effectiveness of the watcher process can be measured in terms of how many brute-force memory layout attacks can be executed before the attack is stopped. The smaller the percentage of the address space that can be targeted, the more effective the watcher process. If the watcher process cannot prevent the attack before all memory locations have been attempted, then the watcher process fails. In the case of a remote attack over a network, Equation 1 describes the number of attacks that can be executed.

$$Exploits = \frac{Bandwidth * T_{response}}{Size_{exploit}} \quad (1)$$

Note that Equation 1 assumes that an infinite number of threads are available to attack, there is no network latency, and that the CPU time consumed by the victim in processing the attack is negligible. While this does not reflect a real-world scenario, it produces a worst-case value of *Exploits*.

## 5 Implementation

### 5.1 Sample Exploit

The exploit used for testing the address-space offset randomization watcher process and response is the original proof-of-concept (PoC) for the *OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow*[6]. The PoC will give a bash shell prompt with the privileges of the apache process on a target machine if executed against a vulnerable apache / OpenSSL target. The total number of bytes involved in this exploit seen on the wire, including overhead like the victim system ACK' responses, is 6158.

This exploit is not the return-to-libc attack that has been shown to be effective against address-space layout randomization; but rather, a heap overflow that can be prevented with  $W\oplus X$  pages alone. It is, however, representative of just the type of automated exploit (worm) that would implement a brute-force mechanism to compromise ASLR-protected systems. In fact, this exploit was used by the Linux "Slapper" worm in 2002 [7]. In the implementation tested, both this heap overflow and a return-to-libc brute-force exploit cause memory exceptions similar enough to be interchangeable for the purposes of monitoring, identification, and response. The timing of the response, and how that timing impacts the potential for memory object location discovery, are the variables being analyzed. It has already been shown that this approach works for ASLR-protected systems[8], so it can be assumed that were this exploit a return-to-libc exploit, it would be successful with proper memory offset values discovered via brute-force.

### 5.2 Test System

The test system used was Fedora Core 1 running the Linux 2.4.32 kernel. As of the writing of this paper, such a system is outdated. However, it is representative of a system that would

have been vulnerable to the OpenSSL vulnerability, and as such is a good candidate. Two versions of the kernel were used: the original, unprotected kernel, and another compilation of the kernel with ASLR and  $W\oplus X$  protections in place. The same system, running the appropriate kernel when necessary, was used for all tests.

The hardware was simulated by VMWare Server Beta 2. This gave a snapshot capability to the tester, ensuring that the same pristine system could be used for every test by falling back to a known-good snapshot. All exploits were performed on an isolated virtual 100Mb network consisting of only the victim machine and the attacking machine.

Apache 1.3.20, with mod ssl 2.8.1 and OpenSSL 0.9.6 were installed on the target system. Again, these versions are badly outdated as of the writing of this paper, but were common when the OpenSSL vulnerability was discovered, and as such made good candidates for our testing.

### 5.3 grsecurity

Grsecurity/PaX's implementation of ASLR and  $W\oplus X$  memory pages was used, without their built-in brute force discovery protections. It is a common realization of ASLR, and provides an extensive set of other tools for protecting Linux systems. It is also often criticized as vulnerable to the return-to-libc attacks mentioned above. As shown by Shacham *et al.*[1], the number of different memory locations an object can reside in is  $2^{16}$ , requiring on average  $2^{15}$  attempts to locate the correct offset of a function on a 32-bit system.

The grsecurity patch 2.1.8, the most recent version available as of the writing of this paper for the Linux 2.4.32 kernel, was used on the protected kernel. Tables in Appendix A list the security settings compiled into the protected kernel, as recommended by the grsecurity documentation[9].

It should be noted that, while the logging settings were sufficient for testing purposes, they should be adjusted to accommodate for a system under attack from multiple IP addresses simultaneously, otherwise, all attacks may not be logged.

The grsecurity patch provides verbose logging. When the OpenSSL exploit was attempted remotely against a protected system, detailed information was recorded in syslog about the event, as illustrated in Appendix D. The information recorded in the system log is more than sufficient to identify and block the offender, including IP address, time, and process ID.

## 5.4 Watcher Process

The availability of source IP address information provided by PaX in syslog meant that monitoring syslog for these kinds of events was all that was required to identify attacks, and attack sources. Once an attack source was identified, it became trivial to block the IP address using the iptables kernel-level IP filtering. This was implemented in a mere 58 lines of Perl code, listed in Appendix B. The IP block instituted was a REJECT rather than a DROP, to assist in measuring the timing of the block by generating ICMP error messages. In practice, a DROP is the preferred response.

A few considerations need to be made when analyzing this watcher process as a solution. First, it is assumed that the target host is not behind a NAT, and that the IP address as seen by the target is the true IP of the attacker. For TCP services monitored by this process, it is unlikely that a remote attacker would be able to spoof the entire TCP handshake, meaning a denial of service via spoofed IP addresses is unlikely. However, for other stateless protocols, this is a real possibility that should be considered. None of these limiting factors were present in the test environment, and did not impact test results.

## 5.5 Findings

Testing showed that the exploit worked on the unprotected kernel, and that the protected kernel successfully prevented exploitation of the vulnerable Apache process. The timing of the response of the watcher process was measured by the elapsed time between the first TCP PUSH packet containing exploit code and the first occurrence of an ICMP type 3 code 9 (administratively prohibited) packet generated by the iptables REJECT entry. The attacking system flooded (ping -f) the target system with ICMP type 8 (echo request) packets while the attack took place. This ensured that as soon as the block was implemented, ICMP administratively prohibited packets were seen from the target back to the attacking host. Table 1 shows the timing results of this test. The average response time for all 10 tests was 540.7665ms with a standard deviation of 220.2321ms.

Table 1: Response time of watcher process

Test	First PSH	ICMP Deny	Delta
0	22:41:04.266882	22:41:04.975192	00:00.70831
1	21:23:41.394067	21:23:41.700986	00:00.306919
2	21:42:08.340528	21:42:08.838062	00:00.497534
3	21:48:27.692289	21:48:28.327417	00:00.635128
4	21:54:35.622568	21:54:36.601139	00:00.978571
5	22:02:28.423513	22:02:28.934483	00:00.51097
6	22:09:16.288196	22:09:16.879952	00:00.591756
7	22:12:52.775387	22:12:53.461324	00:00.685937
8	22:26:21.991759	22:26:22.268045	00:00.276286
9	22:31:14.325933	22:31:14.542187	00:00.216254

## 6 Conclusion

A simple Perl script can detect and respond to a brute-force attempt to discover objects in a randomized memory address space in a half second, on average. Table 2 illustrates that this response time is more than adequate to eliminate the pos-

sibility of brute-force discovery of functions in memory. Thus, with the addition of this simple watcher process to the implementation, address-space randomization achieves its goal of effective prevention of arbitrary code execution for most common network services.

Table 2: Common link speeds and effectiveness of watcher process

<b>Link</b>	<b>Speed (Mb/s)</b>	<b>Exploits Before Block</b>	<b>% of Randomized Address Space</b>
T-1	1.544	16	0.02
T-3	44.736	490	0.75
OC-3	155.52	1704	2.6

## References

- [1] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM Press.
- [2] Brad Spengler. Pax: The guaranteed end of arbitrary code execution. In *G-Con 2: Mexico City, Mexico*, 2003.
- [3] Grsecurity website. <http://www.grsecurity.net/index.php>.
- [4] The PaX Team. Documentation for the pax project. <http://pax.grsecurity.net/docs/>.
- [5] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, 2003.
- [6] John McDonald et al. Bugtraq id 5363. <http://www.securityfocus.com/bid/5363>.
- [7] CERT Coordination Center. Cert advisory ca-2002-027 apache mod ssl worm. <http://www.cert.org/advisories/ca-2002-27.html>.
- [8] Tyler Durden. Bypassing pax aslr protection. *Phrack Magazine*, 59, 2002.
- [9] Brad Spengler. Grsecurity acl documentation v1.5. <http://www.grsecurity.net/gracldoc.htm>.

# Appendices

## A grsecurity "make menuconfig" options

Table 3: PaX Control Options Selected

<b>Parameter</b>	<b>Setting</b>
Support Soft Mode	On
Use ELF program header marking	On
MAC System Integration	direct

Table 4: Address Space Protection Options Selected

<b>Parameter</b>	<b>Setting</b>
Enforce Non-executable pages	On
Segmentation based on non-executable pages	On
Emulate Trampolines	Off
Restrict mprotect()	On
Disallow ELF text relocations	Off
Address Space Layout Randomization	On
Randomize user stack base	On
Randomize mmap() base	On
Deny writing to /dev/kmem, /dev/mem, and /dev/port	On
Disable privileged I/O	Off
Remove addresses from /proc/pid/[maps — stat]	On
Deter exploit bruteforcing	Off
Runtime module disabling	Off
Hide kernel symbols	Off

Table 5: Address Space Protection Options Selected

<b>Parameter</b>	<b>Setting</b>
Hide kernel processes	On
Maximum tries before password lockout	3
Time to wait after max password tries	30

Table 6: Filesystem Protection Options Selected

	<b>Parameter</b>	<b>Setting</b>
	Proc Restrictions	On
	Restrict to user only	Off
	Allow special group	On
	GID for special group	1001
	Additional restrictions	On
	Linking restrictions	On
	FIFO restrictions	On
	Chroot jail restrictions	On
	Deny mounts	On
	Deny double-chroots	On
	Deny pivot-root in chroot	On
	Enforce chdir("/") on all chroots	On
	Deny (f)chmod +s	On
	Deny fchdir out of chroot	On
	Deny mknod	On
	Deny shmat() out of chroot	On
Deny access to abstract AF_UNIX sockets out of chroot		On
	Protect outside processes	On
	Restrict priority changes	On
	Deny sysctl writes in chroot	On
	Capability restrictions within chroot	On

Table 7: Kernel Auditing Options Selected

	<b>Parameter</b>	<b>Setting</b>
	Single group for auditing	Off
	Exec logging	Off
	Resource logging	On
	Log execs within chroot	Off
	Chdir logging	Off
	(Un)Mount Logging	Off
	IPC Logging	Off
	Signal logging	On
	Fork failure logging	On
	Time change logging	On
	/proc/pid/ipaddr support	Off
	ELF text relocations logging	Off

Table 8: Executable Protection Options Selected

	<b>Parameter</b>	<b>Setting</b>
	Enforce RLIMIT_NPROC on execs	On
	Destroy unused shared memory	Off
	Dmesg(8) restriction	On
	Randomized PIDs	On
	Trusted path execution	Off

Table 9: Network Protection Options Selected

	<b>Parameter</b>	<b>Setting</b>
	Larger entropy pools	On
	Randomized TCP source ports	On
	Socket restrictions	Off

Table 10: Sysctl Support Options Selected

	<b>Parameter</b>	<b>Setting</b>
	Sysctl support	Off

Table 11: Logging Options Selected

	<b>Parameter</b>	<b>Setting</b>
	Seconds in between log messages (minimum)	10
	Number of messages in a burst (maximum)	4

## B Watcher Process Perl Script

```
#!/usr/bin/perl
#logwatch.pl
#
# Michael Cloppert
# 4/29/2006
#
# Description:
# Process to monitor syslog for remote PAX violations
# Blocks offending IP address using iptables
#

# User-defined constants
$SYSLOG_F="/var/log/messages";
$TAIL_F="/usr/bin/tail";
$IPTABLES_F="/sbin/iptables";
$IPTABLES_TABLE="RH-Firewall-1-INPUT";

# First, we need to figure out which IP's are already being dropped
# Put into array for easy searching later.
open (IPTABLES, "$IPTABLES_F -nL $IPTABLES_TABLE |");
my @blocklist;
while (<IPTABLES>) {
    if ( /^REJECT/ ) {
        my @line = split ();
        # Source is 4th column
        push(@blocklist, $line[3]);
    }
}
close (IPTABLES);

open (SYSLOG, "$TAIL_F -f $SYSLOG_F |") || \
die("Cannot open syslog file '$SYSLOG_F'\nError was '$!'\n");

while (<SYSLOG>) {
    @fields=split (/:/);
    if (( $fields[3] eq "_PAX" && /execution attempt/ ) || \
        ( $fields[3] eq "_grsec" && /signal 11/)) {
        # Someone's not playing by the rules!
        # Grab IP from the message field
        # get rid of colons
        s/\:\/g ;
        @line=split ();
        foreach (@line) {
            if ( /[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+/ ) {
```

```

# Found an IP. See if it's in the list.
="$-";
my $found=0;
foreach (@blocklist) {
    if ( /$src_ip/ ) { $found++ ; }
}
if (! $found) {
    # Not blocking this IP, start!
    push(@blocklist, $src_ip);
    system("$IPTABLES_F-I_$IPTABLES_TABLE_1-s-$src_ip-j_REJECT");
}
else { die "Exploit_found_from_blocked_IP!\n" ; }
}
}
}
}

```

## C PoC Attacks against System

### C.1 Unrotected System

```
[cloppemj@files bid_5363-apache_mod_ssl]$ ./OpenFuckV2 0x45 fc1
```

```

Establishing SSL connection
cipher: 0x8164308 ciphers: 0x83dbc68
Ready to send shellcode
Spawning shell...
bash: no job control in this shell
bash-2.05b$ bash-2.05b$ --10:29:20--
http://packetstormsecurity.nl/0304-exploits/ptrace-kmod.c
=> 'ptrace-kmod.c'
Resolving packetstormsecurity.nl... failed: Host not found.
gcc: ptrace-kmod.c: No such file or directory
gcc: no input files
rm: cannot remove 'ptrace-kmod.c': No such file or directory
bash: ./p: No such file or directory
bash-2.05b$ bash-2.05b$ whoami
nobody
bash-2.05b$ hostname
fc1-grsec
bash-2.05b$

```

### C.2 Protected System

```
[cloppemj@files bid_5363-apache_mod_ssl]$ ./OpenFuckV2 0x45 fc1
```

```
Establishing SSL connection
cipher: 0x8164308 ciphers: 0x81d5f40
Ready to send shellcode
Spawning shell...
Good Bye!
[cloppemj@files bid_5363-apache_mod_ssl]$ ping fc1
PING vm-128 (172.16.145.128) 56(84) bytes of data.
From vm-128 (172.16.145.128) icmp_seq=0 Destination Port Unreachable
From vm-128 (172.16.145.128) icmp_seq=1 Destination Port Unreachable
```

## D Syslog output of PaX-protected system under attack

```
Apr 28 09:38:56 fc1-grsec kernel: PAX: From 172.16.145.1: execution attempt in:
<anonymous mapping>, 0816f000-08207000 00000000
Apr 28 09:38:56 fc1-grsec kernel: PAX: terminating task:
/usr/local/apache/bin/httpd(httpd):24806, uid/euid: 99/99, PC: 081d65d0, SP:
5a0d4d3c
Apr 28 09:38:56 fc1-grsec kernel: PAX: bytes at PC:
eb 0a 90 90 90 90 90 d0 e8 16 08 31 db 89 e7 8d 77 10 89
Apr 28 09:38:56 fc1-grsec kernel: PAX: bytes at SP-4:
00000002 080da2c7 081de688 00000000 00001078 00000000 081d0310
0816e6b8 081d0310 080c6e83 081de688 00000000 000000c8 080c6ddf
00000000 0816e6b8 081d0310 080c49a6 081d0310 081d0310 081d03e8
```