Improving address space randomization with a dynamic offset randomization technique

Conference Paper · April 2006

DOI: 10.1145/1141277.1141364 · Source: DBLP

2 authors, including:

Steve Chapin
Lawrence Livermore National Laboratory
91 PUBLICATIONS **1,647** CITATIONS

SEE PROFILE

Improving Address Space Randomization with a Dynamic Offset Randomization Technique

Haizhi Xu
System Assurance Institute
Syracuse University
Syracuse, NY 13244
hxu02@ecs.syr.edu

Steve J. Chapin
Systems Assurance Institute
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY 13244
chapin@ecs.syr.edu

ABSTRACT

Address Space Randomization (ASR) techniques randomize process layout to prevent attackers from locating target functions. Prior ASR techniques have considered single-target attacks, which succeed if the attacker can locate a single, powerful system library function. These techniques are not sufficient to defend against *chained return-into-lib(c) attacks*, each of which calls a sequence of system library functions in order.

In this paper, we propose a new ASR technique, *code islands*, that randomizes not only the base pointers of memory mapping (*mmapping*), but also relative distances between functions, maximally and dynamically. Our technique can minimize the utility of information gained in early probes of a chained return-into-lib(c) attack, for later stages of that attack. With a pre-defined rerandomization threshold, our code islands technique not only is exponentially more effective than any prior ASR technique in defending against brute-force searches for locations of multiple targets—a key component of chained return-into-lib(c) attacks, but can also maintain high service availability even under attack. Our overhead measurement on some well-known GNU applications shows that it takes less than 0.05 second to load/rerandomize a process with the necessary C system library functions using code islands, and our technique introduces a 3–10% run-time overhead from inter-island control transfers. We conclude that the code island technique is well-suited to dedicated multi-threaded servers.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

SAC'06 April 23-27, 2006, Dijon, France

General Terms

Security, randomization

Keywords

Address space randomization, Derandomization attacks, Denial-of-service attacks, Intrusion mitigation, Code islands

1. INTRODUCTION

Malicious code execution can be achieved either through injected code execution (e.g. shellcode attacks [1]) or through control transfer using code already in the process (e.g. return-into-lib(c) attacks [15]). There are difficulties in preparing shellcode. First, an overflowed buffer may be too small (for example 16 bytes) to accommodate a complete piece of shellcode. Second, the overflow may be due to wide character conversion: for example, the string `''ABCD''` (four bytes) might be converted to wide characters or Unicode, such as `''0A0B0C0D''` (eight bytes), in which case, injected code is subject to the conversion as well [10]. Finally, various techniques have been developed for detecting injected code executions [17, 19]. Due to these restrictions, attackers may turn to return-into-lib(c) techniques.

A return-into-lib(c) attack overwrites a code pointer used by a legal branch instruction such as `ret`. The overwritten pointer may point at a location of the attacker's choice—usually a function entrance point. Return-into-lib(c) attacks do not necessitate code injection, so they can circumvent detection for injected code execution.

Although proof-of-concept return-into-lib(c) exploits usually call only one C library function—`system()`—there are situations in which attacks invoke multiple functions:

1. If a system call sequence monitor [11] is installed in the target system, the attacker has to use a mimicry attack [22] to escape notice by the monitor. A mimicry attack calls one or more legal system calls before calling the real attacking one, to make the whole system-call path "appear" legal to the monitor. For example, a mimicry attack shown in [22] uses 31 different system calls. When a mimicry attack uses the return-into-lib(c) technique, the attacker needs to call multiple system call wrapper functions located in the C system library.

2. A vulnerable application may be forced to drop privileges before executing a powerful function. For example, the program executed by `system()` may not have setuid or setgid privileges, even if the parent program is a setuid root program. `System(<string>)` executes a command by calling `/bin/sh -c <string>`. In systems in which `/bin/sh` is bash version 2, privileges are dropped before executing the command. The unavailability of an interactive root shell forces attackers to call a chain of other functions.

3. An attack, either by its very nature or because the desired single-step attack function is not available, may use a sequence of functions. For example, the `Code Red` worm [9] has functionality that requires multiple system calls (or corresponding system library functions, if implemented as return-into-lib(c)). Typically, they spawn multiple processes or threads to speed up worm propagation, access files to infect them, and open sockets to probe and propagate to other computers in the network. `Code Red` calls 12 system library functions (in Windows) to create threads, create files, obtain system local time, open sockets, etc [9].

A key attribute of a return-into-lib(c) attack is that the attacker needs the location(s) of the target function(s). In a chained return-into-lib(c) attack, the attacker must acquire locations of all target functions on the chain.

Current operating systems and compilers rely on certain process layout conventions, which attackers can use to find target locations. In a Linux system, for example, a program is `mmapped` into the address space starting from address 0x08048000; the base-pointer of the stack is 0xbfffffff; the base pointer of shared libraries is 0x40000000; and all shared libraries, if loaded, are mapped into one segment.

Address-space randomization (ASR) techniques randomize the conventional address space layout to increase the difficulty of finding the locations of the target functions [4, 5, 21, 23]. However, Shacham et al. showed a *derandomization attack* that finds one target-function location by brute force, even if a process is protected by ASR [18].

This paper develops the idea of *dynamic offset randomization*. We propose a technique, called *code islands*, that randomizes the relative distances between functions maximally, and at run time. Our technique ensures that knowledge gained in early probes of a chained return-into-lib(c) attack contributes little to later stages of that attack. Compared with prior ASR techniques, the code island technique is not only one of the most effective prevention techniques against single-target derandomization attacks, but also the most effective technique to deter multiple-target derandomization attacks; in addition, the rerandomization of our code islands is much faster than other systems.

In the next section, we describe our attack model. We summarize related work in section 3. Section 4 describes our design and implementation of code islands. In section 5, we show our case study of applying the code islands technique to the GNU C system library—`glibc`—on a GNU/Linux x86 system. Section 6 compares the efficacy of available ASR techniques against derandomization attacks. Finally, section 7 gives our conclusions. In the rest of this paper, we use the terms *target* and *location of target function* interchangeably.

2. ATTACK MODEL

The goal of this work is to defend against return-into-lib(c) attacks. We assume that complementary security mechanisms prevent attackers from executing shellcode. Just as with prior ASR techniques, we assume that attackers do not have the power of arbitrary memory accesses.

Return-into-lib(c) is a technique that can be used in buffer overflow exploits or any other control flow redirection attack. It overwrites code pointer(s) with the location of a library function or any other function in the process space that can achieve the attacker's purposes. Because code pointers may be located in any of the data section, the heap, or the stack, it is difficult to prevent or monitor code pointer manipulations efficiently. We assume that attackers can modify one or more code pointers in a vulnerable process (e.g., through buffer overflow attacks).

In a return-into-lib(c) attack that calls multiple target functions, the attacker needs to provide locations of the target functions, proper parameters, and a way to chain these function calls. For example, on the stack shown in figure 2, the attacker needs to provide locations of function `f1()` and `f2()`, two target functions, to exploit a vulnerable binary program compiled with `-fomit-frame-pointer`. Using an `esp lifting` method, the attack first overwrites the return address of a vulnerable function by `f1`, which returns to `eplg`, pointer of an epilogue, which adjusts the stack pointer to the location storing `f2`; then the program jumps to `f2` by executing the instruction `ret` [15].

In a process protected by the ASR techniques, one important method to obtain target-function locations is using a *derandomization attack*, in which the attacker searches for targets by brute force using all available process layout information.

Each trial of a derandomization attack is a remote (chained) return-into-lib(c) attack that uses guessed target(s). To launch a chained return-into-libc attack successfully, the attacker must guess all targets correctly in one trial.

The attacker may use a smart strategy to reduce the number of trials. For example, if the attacker knows the size of the program and where the program is located, then he needs to search only within these bounds.

The attacker may observe the output to see whether a trial succeeds or not. Events that are observable from remote sites are delay of response, no response within a defined period of time, or a corruption of the TCP connection that is used in sending the attack packet(s). If an attacker tries multiple targets in one trial (by using a chained return-into-lib(c) attack), all he can observe remotely is whether or not all targets were guessed correctly.

3. RELATED WORK

the following process layout convention: a process has several segments and every segment is a contiguous block in the virtual address space, so the ordering of the functions and their offsets are fixed within a library. For example, the virtual address of a `libc` function in a system can be represented as

$$address = 0x40000000 + offset + delta\_mmap \qquad (1)$$

where *offset* is the relative distance between the base pointer of `libc` and the location of the function, and *delta_mmap* is a shift from the well-known library base location $0x40000000$

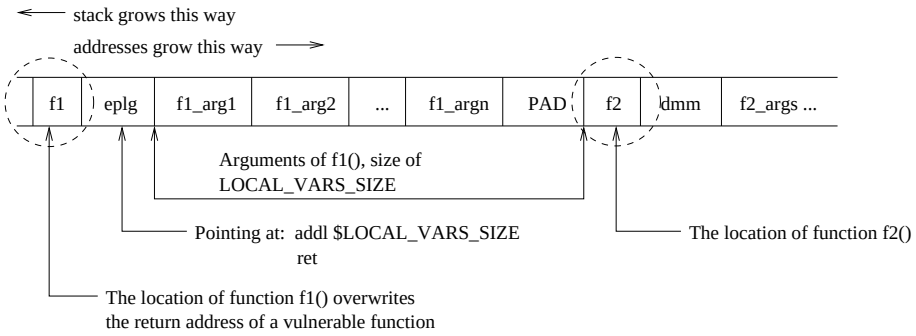

Figure 1: Esp lifting: one method of chaining return-into-lib(c) calls

to the real base location of `libc`[1].

PaX ASLR [21] and transparent runtime randomization (TRR) [23] randomly shift segment base pointers, i.e. the *delta_mmap* in formula (1). Both of them randomize the base pointers at program load time; while PaX ASLR is a kernel patch that modifies the ELF format handler, TRR modifies the dynamic loader. We classify these techniques as *base-pointer randomization*.

Base-pointer randomization prevents attackers from finding the first target easily. Current implementations of base-pointer randomization introduce 16 bits of entropy on a 32-bit architecture, leaving the remaining 12 bits for page offset and 4 bits for the system convention.

However, base-pointer randomization is not sufficient in defending against dedicated attackers. First, once the attacker finds the absolute address of one function, he can deduce the address of any other function by their relative distances. Durden demonstrated how to use knowledge of in-library relative distance to find new targets from information leakage on the user stack [8]. Second, Shacham et al. demonstrated that a brute-force derandomization attack can successfully break into a vulnerable program in a system protected by 16-bit base-pointer randomization in a few minutes [18].

Bhatkar and his colleagues proposed an address obfuscation technique that can randomize the relative distances between functions, i.e., the *offset* in formula (1), and add a random gap between two adjacent functions [4]. In this scheme, a permutation of functions and variables and a random padding between functions are performed at compile and link time. We call this a *static offset randomization* technique. Address obfuscation can be improved in the following aspects. First, the randomization space for permutation is bounded by the size of the program. Second, the permutation is performed statically. As our analysis in section 6 shows, if no rerandomization is performed when a vulnerable program is under attack, effort to find multiple targets is linear to that of finding one target; if a rerandomization (i.e. re-compiling and linking the program in the implementation of address obfuscation) is performed between two attacks, the computation cost of a procedure of static offset randomization is high. For example, it takes us 30 minutes to compile and link the whole GNU libc in a 2.0GHZ Pentium IV system with 1GB memory.

One can combine intrusion detection and ASR techniques to slow down, analyze, or counteract the derandomization attack. For example, an intensive brute force attack can generate a noticeable intrusion signature—frequent segmentation faults. The operating system kernel can detect such brute force attacks by monitoring processes on their frequency of segmentation faults. `Segvguard` is an in-kernel monitor which intercepts partial exception handlers (e.g. SIGSEGV, SIGKILL, SIGBUS, and SIGILL) [16]. Once deciding that a process is being attacked, `Segvguard` will change the state of the process to `TASK_UNINTERRUPTIBLE`—a sleeping state that cannot be woken up by a signal or a timer. Further actions include terminating the process or banning it from calling `setuid`, `fork`, and `execve`. A weak point of `Segvguard` is that attackers may use this feature to launch a denial-of-service attack. Liang et al. proposed an approach that correlates memory errors and generation of attack signatures [13]. Automatic intrusion signature generation from memory errors is an interesting idea [14, 24]. The signatures can be immediately put into a database to filter newly arriving packets. It is not clear yet whether or not attackers can use this mechanism to block legal packets.

There are several other system randomization techniques available. Pointguard encrypts pointers stored in memory and decrypts them when they are loaded into the CPU registers [6]. Instruction set randomization runs the program on a virtual CPU [2, 12]. A recent study has shown that an implementation of the instruction set randomization technique can be broken by using an incremental key-breaking brute force attack [20]. System call renumbering encrypts the system call numbers, which in effect creates a virtual OS API for each process. These techniques target injected code execution, but do not prevent return-into-lib(c) attacks.

4. DESIGN AND IMPLEMENTATION

Code islands break the conventional assumption that content of multiple shared objects be loaded into a contiguous segment. We partition a program into small blocks (we compare them to islands), each containing only a fraction of the original program, then randomly map them into the address space (we compare the address space to the sea) at program load time. While techniques we present could be applied at a granularity as fine as a basic block, a function is the natural granularity for an island, because it is the common unit both for resolving exported symbols and for use in return-into-lib(c) attacks.

4.1 Principles in Island Formation

Our principles in island formation are: (1) split the pro-

[1]This notation comes from [18].

gram as much as possible, such that from one island attackers can deduce little layout information of code and data on other islands. (2) to reduce the overhead, small islands may be merged into one island, as long as the merging does not compromise security.

We first place every function of the program to be transformed in a separate island. Then we optimize the islands by merging islands and duplicating functions into their caller islands, as long as security is not compromised. For example, if function A is a leaf node called by multiple functions, and it is not security critical, then function A can be duplicated into its caller islands, in order to reduce the number of inter-island control transfers. Deciding which functions are security critical is a heuristic step. We consider all system call wrapper functions and all other application binary interface (ABI) functions as security-critical functions. Additionally, if there are two functions with the same symbol name but different version numbers, we merge them into one island. We assume these functions perform similar functionality. Finding all of them does not increase an attacker's power beyond finding only one of them.

4.2 Generating Islands and Mapping Them Into an Address Space

We generate islands from the C source code, following the principles in section 4.1. We require that the program to be transformed follows structured programming guidelines.

We compile every island into a separate shared library. Accesses to code and data on different islands use the standard ABI interface. This implementation has a few advantages. First, the procedure linkage table (PLT) and global offset table (GOT) can be distributed into islands and the tables will be mapped randomly into a process space along with the islands. As we further discuss in section 4.5, distributing the PLT and GOT tables can reduce information disclosure. Second, we make full use of the ELF format, instead of creating a new format, as the ELF format is widely supported by both hardware architectures and software systems.

A program may consist of thousands of functions, resulting in thousands of shared libraries. To save the numerous `open`s when loading these libraries, we use a special archive to hold the islands. In addition to a header that describes islands in the archive, islands are page aligned in the archive because the granularity of mmapping an island is one page.

After island generation, we have a choice of linking in all islands from the original library, or only those that are actually used in the target program. We prefer the latter, thereby enforcing the Principle of Least Privilege that is violated by using a conventional C library. In the conventional C library, as long as one function or variable is needed by a process, the whole C library will be mapped into the process. When we selectively use a subset of the islands, we must also link in any other island that the subset depends on.

We have modified the dynamic linker to support island mapping. At program load time, islands are `mmapped` into the address space at random unoccupied locations. To provide the maximum entropy, a uniform distribution pseudo-random number generator should be used for generating the random locations for the islands.

4.3 Speeding Up Rerandomization

The relocations can be divided into relative relocations and symbol relocations. A relative relocation is intra-object—relative to a load-time available base pointer (e.g. the base pointer of section `.text`); while a symbol relocation is inter-object—the loader needs to find which library contains this symbol.

The relocation algorithm in the current dynamic loader is not efficient when the number of binary objects (mostly shared libraries) in the process is large. In island layout, the number of binary objects can increase from the order of 1 to the order of 10^3, depending on the number of functions in the original program to be transformed. As we will show in section 5.2, it takes the original relocation algorithm 17-20 seconds to load GNU libc in code islands.

To speedup the load/rerandomization procedure, we have developed a new relocation algorithm in the dynamic loader. We built a global hash table for all symbols in all objects in the process at program load time. Using a global hash table can avoid searching the symbol in all dependent objects and therefore reduce the time of finding one symbol from O(n) to O(1) time. We will give performance and analysis of our new dynamic loader in section 5.2.

4.4 Inter-island Control Transfers

In an address space with islands deployed, the caller and called functions might not be located on the same island. Exported symbols are resolved at program load time (*eager relocation*) or postponed until use time (*lazy relocation*). The caller needs a symbol resolver (e.g. `dl_lookup_symbol()`) to retrieve the location of the callee by its symbol name.

An attacker may use the resolver to find the location of any exported symbol and then jump there [15]. To preventing attackers from using the symbol resolvers, we force all exported symbols to be resolved at load time. This eager relocation procedure can be achieved by setting the environment variable `LD_BIND_NOW=1` when running a program. After resolving all symbols, we do not need symbol lookup functions, so we can disable them before transferring control to `main()`. This step can be achieved by changing the permissions for the page containing these functions to be non-readable using `mprotect`, or by unmapping the resolver island, if the resolver functions are compiled as one island. Due to the eager relocation, our code island technique does not support dynamically loaded libraries.

4.5 Consideration of Information Disclosure

Although code islands are not designed for removing information disclosure, they reduce the value of disclosed information—finding a target on one island does not necessarily contribute much layout information to finding other arbitrary functions in the address space.

The PLT and GOT may be targets of attack, because the tables are centralized places for storing locations of potential targets. The PLT and GOT are jump tables to targets that are located in other files. Each ELF binary, either an application program or a library, possesses its own PLT and GOT tables. The GOT stores locations of all exported variables and functions after resolution. When a program is transformed into islands, the PLT and GOT are well distributed into islands. Because islands are mapped randomly, attackers can obtain only limited process layout information from finding the PLT or GOT on one island.

Data or garbage on the stack may contain valuable in-

formation for attackers. For example, the return address of `main()` is stored on stack throughout the lifetime of the application program. Using the address, an attacker can deduce the location of at least one function—`__libc_start_main()`, the caller of function `main()`. The attacker may also scavenge the garbage remained on the stack for useful code pointers. To reduce the information leakage from the stack, we suggest called functions clean up their local frames before returning to their caller functions and callers erase the return addresses from the stack immediately after returning.

4.6 Rerandomization

Redeployment of address space randomization is called *rerandomization*. Prior work shows that rerandomization adds only one bit of entropy against a brute-force attack guessing the location of a single target function [18]. However, if an attacker searches for *multiple* targets, especially if he has obtained some targets and he tries to find new targets using the obtained information, then rerandomization becomes an effective defensive technique. We show in section 6 that code island layout with a threshold-based rerandomization is exponentially better than prior ASR techniques in blocking multi-target derandomization attacks.

With high probability, a random guess on a target address will result in a segmentation fault. When the processor fetches an instruction from an unmapped address or from an address containing an illegal opcode, an exception is raised by the processor. If the target victim is a process, the default exception handler will kill the process. After the program is reloaded, the address space randomization may be redeployed.

In a multithreaded process, when a segmentation fault happens in a thread, the process may kill that thread and spawn a new thread, leaving the layout of the address space unchanged. As long as the process is not reloaded, the knowledge that the attacker has obtained is not lost. What is worse, if the target system is a commercial server, it can neither afford to ignore the attack nor the cost of rerandomization after each failed trial.

One solution is to force multi-threaded processes to restart after the number of segmentation faults reach a pre-defined threshold, which is set based on statistics of the total number and frequency of segmentation faults. Once the threshold is reached, the exception handler forces the process to terminate and the program to restart. After restart, the address space is rerandomized and prior layout information becomes outdated. Although restarting may force the process to lose its context, most transaction-related information can be stored in persistent storage. The program restarting approach differs from `segvguard` [16]. In `segvguard`, after segmentation faults reach a threshold, the system will terminate the process or disallow forking and execing by the process, turning the attack into a denial-of-service attack. A proper threshold value can both prevent derandomization attacks and maintain a high system availability, as we show in section 6.

Restarting a program is a fast rerandomization technique, compared to the time cost of recompilation in static offset randomization techniques. When applying the code island technique to a dedicated server, the restart time overhead will be amortized over the lifetime of the server process.

5. EXPERIMENT WITH THE GNU LIBC

In this section, we present our results from applying the code island technique to the GNU C system library—`glibc`, which contains wrappers of system service requests and other basic system facilities. Binary applications on a GNU/Linux machine link against either the static version or the shared version of the C system library. Randomizing the layout of the C library makes it difficult for return-into-lib(c) attacks to find the locations of system call wrappers.

GNU libc version 2.3.2 contains 2876 functions. Following the principles in section 4.1, we obtained 2708 islands. Each libc island was compiled as a separate shared library. When linking a program, we linked the program against the libc islands, rather than the original libc, by passing the path where libc islands was located and names of the islands to `ld`, the GNU linker, as the following compile command illustrates. The compiler flags in italics caused the linker to use libc islands.

```
[user@duallcd barcode-0.98] make
gcc -g -O2 -Wall -DHAVE_UNISTD_H=1 -o barcode
-DHAVE_STRCASECMP=1 main.o cmdline.o -DNO_LIBPAPER -L.
-lbarcode -L../../islandslibc -Wl,-lic__nl_normalize_code
set, -lic__IO_free_wbackup_area, -lic__gconv_translit_find,
...
```

In the rest of this section, we show the temporal and spatial overhead of our current implementation of the GNU C system library in the code island layout.

5.1 Spatial Overhead

As a shared library, every island on disk needs an i-node and a data block. The code section and read-only data section of a shared library can be shared with other processes, so in memory there is only one copy of code and read-only data of an island. However, the data section is copy-on-write, so there may be multiple copies of the data section of an island in a system. In addition, each section is page-aligned. If one section of an island is smaller than a page, which is the common situation, the remaining fragment of the page will be wasted.

We compare memory usage of a process with glibc-2.3.2 using both conventional and code island layout. The conventional libc code occupies 302 pages, or approximately 1.2MB of memory. If all libc islands are mapped in, island libc code occupies around 10.6MB memory. Because islands are shared libraries, the code sections bear a per-system, not per-process, overhead. The data section of the original libc occupies 4 pages. In the code island layout, however, the data section of each island occupies at least one page and the data section is not shared with other processes. In addition to the overhead from code and data sections, each island, as s shared library, requires a VM structure for memory management per system.

Unlike the conventional layout, where a whole library must be mapped in the process as long as one function of the library is called, the code island approach can map in only necessary islands to the program. A few GNU applications programs that we studied require between 254 and 555 islands, only 8.8–19.3% of libc code. According to the classification of Bernaschi, et al. in [3], there are 22 dangerous system calls in GNU/Linux, which are classified as ranking at threat level 1. Not all of these dangerous system calls are used by application programs: at most 9 out of the 22 were used in the application programs we studied. Every applica-

program	load time, conventional layout libc, lazy relocation	load time, libc in islands, eager relocation			
		all libc islands loaded		only necessary libc islands loaded	
		new relocation algorithm	original relocation algorithm	new relocation algorithm	original relocation algorithm
a2ps	0.001s	2.561s	11.9s	0.027s	0.180s
aspell	0.008s	5.509s	19.7s	0.032s	0.615s
barcode	0.001s	5.410s	17.3s	0.015s	0.126s
calc	0.001s	2.656s	17.1s	0.022s	0.244s
diction	0.001s	2.604s	17.0s	0.024s	0.285s
enscript	0.001s	2.547s	17.3s	0.025s	0.249s
finger	0.001s	2.570s	17.9s	0.048s	0.592s
gv	0.004s	2.607s	17.6s	0.021s	0.222s
http tunnel server	0.001s	2.995s	17.3s	0.023s	0.246s

Table 1: program load/rerandomization time with libc using different layout schemes

tion uses `open()` and 5 programs use `execve()` or `system()`. The numbers tell us that over 80% of libc functions are not used by typical programs, some of which are security critical to the system. Exposing these functions to the application not only is unnecessary, but also facilitates return-into-lib(c) attacks.

5.2 Temporal Overhead from Rerandomization

When islands are in use, the overhead at program load time comprises both the loading overhead and the rerandomization overhead. Besides the time for `execve`, an in-kernel system service routine, program loading can be divided into program mmapping and relocations. During mapping of each shared library, the dynamic linker first interprets the file header, then `mmap`s necessary parts of the library into the address space based on the information stored in the file header. After that, the mapped objects are relocated to their real locations.

When a conventional program is transformed into multiple islands, many internal function calls or data references become exported references. In addition, we require eager relocation of all external symbols at program load time. Therefore, both the program mmapping time and relocation time increase. Table 1 compares the load time of some GNU applications with libc in the two layout schemes.

We measured the load time of programs uing libc with the conventional layout to be approximately 0.001 seconds, of which around 40% is used for relocation (lazy relocation is used as it is the default relocation policy). When code islands are in use, only necessary libc islands are mapped in, and our new relocation algorithm is applied. It takes less than 0.05 seconds to start the programs, which was not noticed by our testers using interactive programs.

The number of islands to be loaded into an address space is another factor of the overhead. When all libc islands are mapped into the process, the number of objects are in the order of 1000 and it takes 2.5-5.5 seconds to load the program using our optimized relocation algorithm. If we load only necessary libc functions, then the number of objects is order 10^2 and the process load time is 0.05 second using our optimized relocation algorithm. The object mapping time does not change much although we use a special archive to store all the islands.

The program load time in table 1 shows that our new relocation algorithm can load a program 6-10 times faster than the original relocation algorithm. The relocation time decreases significantly, from more than 83% of the overall program load time to only 1-3% of the program load time.

The temporal complexity of relocation using our new relocation algorithm is $O(R+N+r)$, where R is the number of relative relocations, N is the number of exported symbols in all shared objects plus the main executable, and r is the number of symbol relocations. Our new symbol relocation algorithm first builds a global hash table containing all exported symbols in all objects in the process address space, which costs $O(N)$ time, then a symbol relocation becomes a search in the global hash table, which costs $O(1)$ for every symbol relocation. The cost of relative relocations is not changed in our new relocation algorithm. The temporal complexity of using the original relocation algorithm is $O(R+nr)$ time, where n is the number of shared objects plus the main executable [7].

After relocation, we can release the global hash table, because we no longer need it throughout the process life time, since we require eager relocation and we do not allow dynamically loaded libraries.

5.3 Run-Time Overhead from Inter-Island Control Transfer

After program loading is complete, the program run-time overhead from code islands is mainly from inter-island control transfers. In our measurement, control transfer between libc islands increases program run time by 3–10%, as shown in figure 2. An intra-library function call is only one `call` instruction, while an inter-library function call jumps to an entry of the PLT table first, then jumps indirectly to the target, the location of which is stored in an entry of the GOT. The overall overhead of inter-island control transfers depends on both per-call overhead and the frequency of the calls.

6. EFFECTIVE ANALYSIS

Our analysis in this section is based on the best attacking strategies that we found, and the results serve as upper-bounds in the schemes under study. Recall that, in our attack model described in section 2, each trial is a (chained) return-into-libc attack using one or multiple hypothetical

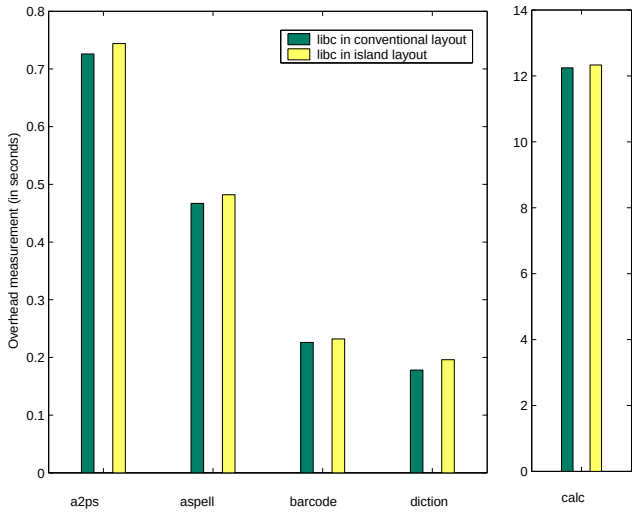

Figure 2: Runtime overhead of control transfer between libc islands

target locations, in addition, the attacker can only observe whether or not all targets are correct.

We assume all randomization schemes use a uniform distribution. Our analysis is based on a 32-bit x86 Linux system. Assume that the size of a page is 2^k bytes, there are 2^n pages of space available for randomization, and the program occupies 2^m pages in memory. In our analysis, $n = 16$, because in a 32-bit address, 12 bits are reserved for the page offset (i.e., 4K pages) and the first 4 bits of the address are used by convention to differentiate segments of the address space; $m = 8.25$ for the example of `glibc`, which is 1.2MB in size.

6.1 Randomization Schemes Against Derandomization Attacks

In this section, we derive the average number of trials for a derandomization attack to succeed under different randomization schemes. Because of space limitations, we cannot include the proofs here.

THEOREM 1. *In all ASR schemes without rerandomization after failed trials, the expected number of trials to find t ($t = 1, 2, 3, \ldots$) targets is linear in the average number of trials for finding one target.*

Under all schemes without rerandomization, no layout change is made after failed trials. An attacker can try one target in a trial and combine all found targets in the last trial.

Let B denote the base-pointer randomization scheme using the granularity of a page [21, 23], F denote the function-and-variable permutation scheme [4], BF denote the scheme combining the base-pointer randomization and function-and-variable permutation, and IC denote the island code scheme.

THEOREM 2. *Let $B_R(t)$, $F_R(t)$, $BF_R(t)$, and $IC_R(t)$ be number of trials for finding t functions under schemes B, F, BF, and IC with rerandomization after each failed trial. To find t ($t = 1, 2, 3, \ldots$) targets under each of the rerandomization schemes, the average number of trials is*

$$E[B_R(t)] = 2^n \quad (2)$$

$$E[F_R(t)] \leq t \cdot 2^{m+11} \quad (3)$$

$$E[BF_R(t)] \leq 2^{n+12} + (t-1) \cdot 2^{n+m+13} \quad (4)$$

$$E[IC_R(t)] \approx 2^{t \cdot n} \quad (5)$$

Among the four schemes analyzed in theorem 2, the code island scheme is the most effective one in defending against multi-target derandomization attacks. Under the code island scheme, the number of trials for finding multiple targets is exponential in the number of targets; while under the other three schemes, the effort of finding multiple targets is linear to that of finding one target.

6.2 Using a Threshold in Rerandomization

If a rerandomization is triggered after every segmentation fault, then attackers may use this feature to launch a denial-of-service attack, keeping the server busy with rerandomization. To prevent such denial-of-service attacks, we introduce a threshold, T, the number of failed trials between rerandomizations. The threshold should be small enough such that the attack has low probability of finding all necessary targets, while large enough such that the server is still highly available even if it is under attack.

Let t be number of targets that an attacker is searching for.

THEOREM 3. *Let $IC_T(t)$ be the number of trials of a derandomization attack to succeed on t targets in space N with a rerandomization threshold T ($T \leq N$), and α be the probability of success in $IC_T(t)$ trials, we have*

$$IC_T(t) \geq [\alpha \cdot t! \cdot (\frac{N}{T})^t - 1] \cdot T \quad (6)$$

Theorem 3 tells that in the code island scheme, when threshold T is fixed, the number of trials for finding all the targets increases exponentially as the number of targets increases. In practice, we let the threshold T be much smaller than N, for example $T/N = 0.001$. Location information obtained by the attacker is only useful for at most T failed trials, when rerandomization occurs.

We define the availability of a server as

$$A = 1 - \frac{\textit{time for a randomization}}{\textit{interval between two successive randomizations}} \quad (7)$$

Under the scheme of code islands with page-offset randomization, the randomization space is $N = 2^{28}$. A rerandomization procedure takes around 0.05 seconds, as our measurement showed in section 5.2. In a TCP-connection based attack, if there are 273 connections per second, which supports up to 22 million connections per day, and assume that all incoming connections are from a derandomization attack, when $T = 0.001N$, the interval between two successive rerandomization is 983 seconds and the server's availability $A = 0.99995$. In this situation, attackers need to try approximately 10^{21} trials to reach probability 0.05 and 10^{23} trials to reach probability 0.95 for finding 5 targets.

7. CONCLUSION

In this paper, we introduce a new address space randomization technique—code islands. Our analysis shows that

the code island technique is the most effective ASR technique available in deterring multi-target derandomization attacks. Rerandomization plays an important role in blocking multi-target attacks, so server programs should be rerandomized after the number of segmentation faults reaches a threshold. Our experiments show that a rerandomization procedure costs less than 0.05 seconds, when an application program loads only necessary libc islands.

The code island technique can be best applied to protecting dedicated multi-threaded servers, which demand both strong protection and high availability. In a server process, the island loading/rerandomization overhead will be amortized over the life time of the process. Memory consumption of islands in a dedicated multi-threaded server is not as high as in general-purpose multi-process systems. In addition, a 3–10% run-time slowdown is a reasonable sacrifice of performance for protection.

8. ACKNOWLEDGMENTS

We thank Dr. Wenliang Du for his discussions and work with the authors. We also thank Dr. Kyungsuk Lhee for his numerous suggestions and comments.

9. REFERENCES

[1] Aleph One. Smashing The Stack For Fun And Profit. *www.Phrack.org*, 49(14), November 1996.

[2] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference On Computer And Communication Security*, October 2003.

[3] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Enhancements to the linux kernel for blocking buffer overflow based attacks. In *4th Linux showcase & conference*, October 2000.

[4] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, Washington D.C., August 2003.

[5] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, August 2005.

[6] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[7] U. Drepper. How to write shared libraries, January 2005. http://people.redhat.com/drepper/dsohowto.pdf.

[8] T. Durden. Bypassing PaX ASLR protection. *www.Phrack.org*, 59(9), June 2002.

[9] eEye Digital Security. ANALYSIS: .ida "Code Red" Worm, July 2001. http://www.eeye.com/html/research/advisories/AL20010717.html.

[10] eEye Digital Security. Microsoft Internet Information Services Remote Buffer Overflow (SYSTEM Level Access), June 2001. http://www.eeye.com/html/Research/Advisories/AD20010618.html.

[11] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.

[12] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference On Computer And Communication Security*, October 2003.

[13] Z. Liang and R. Sekar. Automated, Sub-second Attack Signature Generation: A Basis for Building Self-Protecting Servers. Technical report, Department of Computer Science, Stony Brook University, May 2005.

[14] Z. Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, Alexandria, VA, November 2005.

[15] Nergal. The Advanced Return-into-lib(c) Exploits. *www.Phrack.org*, 58(4), December 2001.

[16] openwall. segvguard. ftp://ftp.pl.openwall.com/misc/segvguard/.

[17] W. Purczynski. kNoX—Implementation of Non-Executable Page Protection Mechanism. http://www.opennet.ru/prog/info/1769.shtml, May 2003.

[18] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address Space Randomization. In *Proceedings of the 11th ACM Conference On Computer And Communication Security*, Washington, DC, USA, October 2004.

[19] Solar Designer. Non-Executable User Stack. http://www.openwall.com/linux/.

[20] N. Sovarel, D. Evans, and N. Paul. Where's the FEEB? The Effectiveness of Instruction Set Randomization. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, August 2005.

[21] the Pax team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[22] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference On Computer And Communication Security*, Washington, DC, USA, November 2002.

[23] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the 22nd Symposium on Reliable and Distributed Systems (SRDS)*, Florence, Italy, October 2003.

[24] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, Alexandria, VA, November 2005.