Transparent Runtime Randomization for Security

Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{junxu, kalbar, iyer}@crhc.uiuc.edu

Abstract

A large class of security attacks exploit software implementation vulnerabilities such as unchecked buffers. This paper proposes Transparent Runtime Randomization (TRR), a generalized approach for protecting against a wide range of security attacks. TRR dynamically and randomly relocates a program's stack, heap, shared libraries, and parts of its runtime control data structures inside the application memory address space. Making a program's memory layout different each time it runs foils the attacker's assumptions about the memory layout of the vulnerable program and makes the determination of critical address values difficult if not impossible. TRR is implemented by changing the Linux dynamic program loader, hence it is transparent to applications. We demonstrate that TRR is effective in defeating real security attacks, including malloc-based heap overflow, integer overflow, and double-free attacks, for which effective prevention mechanisms are yet to emerge. Furthermore, TRR incurs less than 9% program startup overhead and no runtime overhead.

1 Introduction

This paper addresses security vulnerabilities that can lead to Unauthorized Control Information[1] Tampering (UCIT) in a target program. UCIT vulnerabilities include many commonly seen security problems such as buffer overflow, format string, integer overflow, and double-freeing of heap buffer. Attacks exploiting UCIT vulnerabilities share a common pattern: the attacker exploits the vulnerability to change critical control information in the target system so that it points to the attacker's malicious code. These attacks do not depend on a legal user to perform any action, however innocently or unknowingly. In contrast, attacks such as the Trojan horse and worm present the malicious code to the user in a disguised manner and entice the user to execute the code. A quick survey of the 109 CERT[2] security advisories issued over the past four years shows the significance of UCIT vulnerabilities (see Table 1). The first four categories in the table are UCIT vulnerabilities, accounting for nearly 60% of all the CERT advisories: *buffer overflow* attacks exploit unchecked buffer; *format string* attacks exploit unchecked format commands in `printf`-like functions; *double free* attacks exploit bugs or design flaws to release unallocated or already freed heap buffers; *integer overflow* attacks exploit signed integer overflow bugs to overwrite memory contents. The other categories include vulnerabilities such as back-door, denial of service, worms, viruses, weak authentication, or insecure default settings.

Vulnerability	Number	Percentage
buffer overflow	49	44.95%
format string	9	8.26%
double free	2	1.83%
integer overflow	3	2.75%
backdoor/Trojan horse	8	7.34%
denial of service	5	4.59%
others	33	30.28%
Total	109	100.00%

Table 1. CERT Advisories (1999-2002)

An analysis of published security attacks that exploit UCIT vulnerabilities shows that a key element in all such attacks is the attacker's ability to determine the runtime addresses of critical memory data elements used by the target applications. Common targets include server applications such as ftp, web, and secure shell servers. Examples of critical memory data elements include buffers, function pointers, and function return addresses, which are usually 32- or a 64-bit addresses. In theory, correctly guessing the value of a 32- or even 64-bit number representing, for example, a function pointer on the program stack, is difficult. In practice however, an attacker using the following approach can reliably determine these address values. The attacker usually starts by scanning hosts on the network to identify vulnerable applications or operating systems. The attacker can then configure a pilot system and experimentally formulate an attack, e.g., determine the address values for use in a real attack. The approach works because major operating systems have well-known default memory layout schemes that are used in production environments. Typically, the attacker needs only to understand details of a few major applications and operating systems, because there are only a few popular operating systems (e.g., Linux, Solaris, Windows), hardware platforms (e.g., x86, SPARC), and major applications in mass use. For example, almost all Unix/Linux systems use *sendmail* (developed by

[1]Control information includes function pointers, return address, and indirect jump targets in the data, heap, or stack of a program.

[2]The CERT Coordination Center (`http://www.cert.org/`) was established after the Morris Internet worm incident in 1988 and funded by DARPA to coordinate communication among experts during security emergencies and to help prevent future incidents.

the Sendmail Consortium [22]) as the electronic mail clients; also NetCraft reports [17] that the web server market is dominated by Apache and Microsoft IIS, (with 65% and 25% market shares, respectively).

A number of specialized solutions protect a system from attacks that exploit specific types of UCIT vulnerabilities. For example, techniques such as StackGuard and LibSafe [10, 6] defeat stack-smashing-based buffer overflow attacks, and techniques such as FormatGuard [9] protect against format string attacks. While these approaches are effective in protecting a system against the specific attack they focus on, incorporating many individual techniques to defend against a wide range of attacks is nontrivial and often requires resolving conflicting requirements imposed by the different techniques.

This paper proposes *Transparent Runtime Randomization* (TRR), a generalized approach to protect systems against a wide range of security attacks that exploit UCIT vulnerabilities. The TRR technique dynamically and randomly relocates a program's stack, heap, shared libraries, and parts of its runtime control data structures inside the application memory address space. Making a program's memory layout different each time it runs foils the attacker's assumptions about the memory layout of the vulnerable program and makes the determination of critical address values difficult if not impossible. An incorrect address value for a critical memory element causes the target application to crash. Although a crash may not be desirable from reliability and availability perspectives, in the security domain a crash is an acceptable option to the program being hijacked. TRR is implemented by modifying the dynamic program loader, therefore it is *transparent* to the application programs, i.e., existing applications run without any modification or recompilation.

Currently, TRR has been implemented on Linux/IA-32 platforms. It is shown, using published attacks, to be effective, not only against well-studied attacks such as stack buffer overflow and format string, but also against attacks such as malloc-based heap overflow, integer overflow, and double-free, for which effective solutions are yet to emerge. While the effectiveness of TRR is demonstrated using the first four types of vulnerabilities in Table 1, its protection is not limited to these four. In fact, the proposed technique can defeat all attacks that need to correctly determine the runtime address of memory elements in target programs. Performance overhead of TRR is negligible, as TRR-related activities are only run at process initialization time. Process startup overhead measured for a wide range of applications ranges between 2-9%. TRR does *not* introduce any additional overhead once the application process begins execution. The TRR technique is portable across most Unix-based operating systems including Linux, FreeBSD, and Solaris. The key to this portability is the use of ELF [23] as the standard binary executable format, which is common to all.

2 Related Research

Techniques proposed to protect systems from attacks that exploit software implementation vulnerabilities can be broadly divided into two types: *static analysis* and *runtime detection*. Static techniques [12, 14, 16, 25] analyze program source code at compile time to find possible vulnerabilities. Runtime techniques [6, 10, 9, 18, 24] either insert special checking code at compile time or instrument the runtime environment to dynamically detect possible security attacks. Most of these techniques aim at providing protection against specific types of attacks, e.g., StackGuard [10] against stack buffer overflow attacks or FormatGuard [9] against format string attacks.

Use of diversity has been advocated by many for achieving reliability and security. Deswarte [11] gives a thorough review of how diversity at different levels of software and hardware systems (user and operator level, human-computer interface, application software level, execution level, and hardware or operating system level) have made those systems more reliable and secure. Existing approaches can be broadly divided into two categories: those that aim at tolerating design/implementation bugs to achieve high *system reliability* and those that aim at achieving *system security*. The N-version programming approach [4] aims at improving software reliability by providing tolerance to software design and implementation bugs through multiple independent versions. The N-version programming approach can also prevent security vulnerabilities due to software implementation errors [8]. It is well recognized that this approach is expensive to develop and operate, except for ultra-high dependability environments.

Introducing randomness to prevent unauthorized access is the principle of all cryptosystems. It has also been successfully used in areas such as cyber-watermarking and copyright protection. In [13], Forrest et al. advocate a broad philosophy for secure systems using diversity. The authors argue that all the advantages of uniformity can a become potential weakness because any bug or vulnerability in an application is replicated throughout many machines; by *deliberately introducing diversity*, a system may become more robust and secure. The paper suggests several possibilities for introducing diversity, mainly through *randomized compilation*. A specific extension to the GNU C Compiler, *gcc*, that pads each stack frame by a random amount, is implemented to defeat stack-based buffer overflow attacks.

Pu et al. [21] proposed specialization techniques [20] to increase the diversity in operating system code. They advocate using program *invariants* or *quasi-invariants* to statically or dynamically plug in different versions of code blocks to defeat possible security attacks. However, subsequent research from the authors does not appear to follow the path laid out in [21]. A survey by Bain et al. [5] shows that using heterogeneous hardware platforms, operating systems, and application software can improve system security and survivability. Since many widespread attacks only target specific applications and operating systems, diversity would prevent all systems in a heterogeneous environment from being subverted.

The Address Space Layout Randomization (ASLR) proposed by PaX [19] implements a similar idea to TRR, i.e., both randomize the memory layout of an application. There are, however, several important differences between the two: (1) TRR is implemented entirely in the user-space dynamic program loader,

while ASLR requires changes to the Linux kernel. User-space implementation does not require re-installation or even reboot of the operating system and hence is easier to use and deploy. (2) TRR randomizes the location of the global offset table (GOT), a frequent target of many attacks, while ASLR does not. The challenge in randomizing GOT is that it is location-dependent and references to GOT must be preserved. (3) TRR and ASLR randomize the locations of heap and shared library differently. TRR randomizes the location of the heap by randomly growing the base of the heap and the location of the shared libraries by inserting a random-sized memory region before the libraries are loaded. In ASLR, both heap and shared libraries are randomized by changing the `do_mmap` system call. Whenever the heap is expanded, a new shared library is loaded or a memory region is mapped, and a random offset is applied. This approach may result in more virtual memory fragmentations, since different memory regions are mapped non-contiguously with random gaps between them. (4) While our study shows that TRR has a small overhead only at process initialization time, the performance impact of ASLR is yet to be evaluated.

3 Common Characteristic of Attacks Exploiting UCIT Vulnerabilities

Recall from Table 1 that attacks exploiting UCIT vulnerabilities account for nearly 60% of known attacks. The *modus operandi* in all these attacks is the same: an intruder launches an attack by sending malicious messages/data to the system running the vulnerable application(s). Messages or data here broadly mean various forms of external inputs that an application can receive, e.g., network message, console input, command line options, or environment variables. Using the malicious messages/data, the attacker attempts to satisfy two conditions: (1) inject malicious code and (2) change existing control information (e.g., return addresses and function pointers) to point to the malicious code. Let m be the address at which the malicious code is to be placed, and let p be the address of the target application's control information. The goal of the attacker is to overwrite the value at p so that the control information now points to the memory address m, where the malicious code is located. Observe that a common characteristic of these attacks is: *in order to achieve the above goal, an intruder must correctly determine the runtime values of m or p or both.* This is usually achieved in the following way: (i) identify the versions of the application and operating system; (ii) configure a pilot system to mimic the target system; and (iii) craft and test the attack using the pilot system. A successful test-run allows the attacker to obtain the values of p and/or m. Table 2 shows a list of representative attacks, the vulnerabilities they exploit, the location (m) where the malicious code is to be placed, the control information at address p that needs to be changed, and the values that need to be determined for a successful attack. Observe that in a stack-smashing attack, the intruder only needs to determine one address, either m or p, while in others such as malloc-based heap attack or integer overflow attack, the attacker needs to determine both m and p.

We propose a generic, randomization-based solution against attacks that exploit this common feature regardless of how the feature is exploited. The proposed technique is shown to provide a robust solution not only for the well-known stack buffer overflow and format string attacks, but also for the others such as malloc-based heap overflow, integer overflow, and double-free attacks, for which effective solutions are yet to emerge. We first explain three representative attacks that exploit stack buffer overflow, malloc-based heap overflow, and integer overflow vulnerabilities.

3.1 Example of A Stack Buffer Overflow Attack

Buffer overflow is the most common attack on the Internet. The data in Table 1 show that it alone accounts for almost half the security advisories reported by CERT over the past four years. Buffer overflow attacks exploit unchecked buffers in application programs by sending more data to the application than the allocated buffer space can hold. By writing past the buffer boundary, control information following the buffer can be maliciously altered. Many variations of buffer overflow attacks exist: some examples are stack smashing [2], return-to-library [26], and malloc-based heap exploits [3, 15].

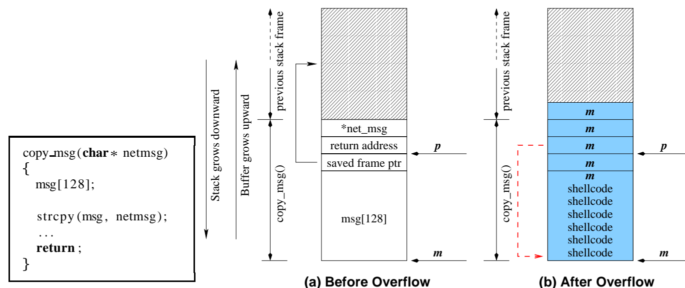


Figure 1. Example of Stack Smashing

Figure 1 shows an example of a stack-smashing attack. Stack smashing [2] exploits unchecked buffers on the program stack and aims at overwriting the function return address for `copy_msg()` at address p following the unchecked buffer `msg[128]` at address m (Figure 1a). The attacker sends a malicious message consisting of two parts: the malicious shell code in the first part, and the new value (in this case m) to overwrite the return address in the second part. Because the program blindly copies `net_msg` to `msg[128]` using the vulnerable function `strcpy` without boundary checking, the shell code is placed at m and the return address at p is changed to point to m (Figure 1b). When `copy_msg()` returns, it transfers control to the malicious code at m, and the attacker takes over. Observe that the attacker needs to determine the runtime value of m (the location of the buffer `msg` where the malicious code is to be placed). Clearly, an erroneous choice of either m or p will cause the function to return to an unknown location, and the program will crash.

[3]Here the attacker needs to change p to point to a function in a program's shared library and force that function to use the malicious data at m; therefore, the attacker needs to determine both m and the address of the library function.

Attack	Vulnerability	m	p	Value to be determined
stack smashing	stack buffer overflow	stack buffer	RA	m or p
ret-to-library	stack buffer overflow	stack buffer	RA	m & p[3]
malloc-heap	heap buffer overflow	heap buffer	FP or RA	m & p
format string	format string validation	format string	FP or RA	m & p
integer overflow	signed integer overflow	env. variables	FP or RA	m & p
double free	free unallocated/released memory	heap buffer	FP or RA	m & p

m - location where the malicious code/data is to be placed;
p - location of control information to overwrite; FP - Function pointer; RA - Return address

Table 2. Different Types of Attacks

3.2 Example of Malloc-based Heap Overflow Attack

This section presents an example of malloc-based heap overflow attack, an important new variation of buffer overflow attack. A program's heap is usually managed by the C library functions `malloc` and `free`. The heap is divided into groups of free blocks of similar size, and blocks in each group are organized using a doubly linked list. For efficiency reasons, the forward pointer, `fd`, and backward pointer, `bd`, that maintain the doubly linked lists are stored at the beginning of each free block. An attacker can exploit unchecked heap buffer vulnerabilities to change these pointers and thereby seize control of the program. We illustrate this attacking method using the example shown in Figure 2.

Figure 2a shows three free blocks `A`, `B`, and `C` in a doubly linked list, and the used block `U` (not part of any free list). When block `U` is freed, it is consolidated with the neighboring free block `B`, and `B` is taken out of its current free block list by the following two operations: `(B->fd)->bk=B->bk` (equivalent to `A->bk=C`, since `B->fd` is `A` and `B->bk` is `C`) and `(B->bk)->fd=B->fd` (equivalent to `C->fd=A`). Figure 2b shows the heap after those operations: `U` and `B` are merged into one larger free block[4], and `A` and `C` are directly linked in the list. Going back to the initial state in Figure 2a, the attacker can send malicious messages to overflow buffer `U`, thus overwriting `B->fd` to point to `p` (the address of a function pointer) and overwriting `B->bk` to point to `m` (the location where the malicious code will be placed) (see Figure 2c). In this case, when `U` is freed, `B` is taken out of the doubly linked lists through two pointer operations: `(B->fd)->bk=B->bk` and `(B->bk)->fd=B->fd`. The first operation is equivalent to `p->bk=m`, hence, the function pointer at `p->bk` is changed to m, where the malicious code is placed (see Figure 2d). The next time the function pointer at `p->bk` is used, the malicious code will be executed. The attacker needs to determine the address values m and p in order to seize control of the program.

3.3 Example of a Signed Integer Overflow Attack

Signed integer overflow occurs when a program derives an integer from an external input, directly or indirectly, without adequate validation. Consider the case in which a program requests an integer from *standard input*, converts the input to a number, and then uses the obtained number to index an array. If the number is unchecked, it could go beyond the boundary of the array and cause an unauthorized memory read/write to occur. This case is illustrated in the following example: an integer overflow attack against the Unix mail transport agent *sendmail*.

Sendmail uses the command line option `-Dcategory level` to change the default verbosity level for any of 100 different categories of debugging information (refer to the pseudocode in Figure 3). The levels are stored in the array `tTdvect[100]` in the data region. The program declares `category` as a signed integer and checks the number converted from the command line option (`category`) to make sure it is less than 100. A malicious user can send a very large number as `category`, so large that it is interpreted by the program as a negative binary number that passes the less-than-100 check in Figure 3. When the program executes `tTdvect[ category ] = level`, `category` is negative and `level` is written to memory location `tTvect - category`. By reverse engineering the program, an attacker can figure out the runtime address of `tTdvect`. The attack is accomplished by overwriting a function pointer at location p (of the attacker's choosing) in the program's global offset table and then embedding malicious code in the program's environment variable at address m on the stack. The intruder can then use the command line option `-D(p - tTdvect) m` to invoke the program [5]. The program essentially executes `tTdvect [ p - tTdvect ] = m`, i.e., `*p = m`, thus changing the function pointer at p to point to m. The next time the function pointer at p is used, the program is hijacked. Once again, the attacker needs to determine the runtime values of p (the address of the function pointer to overwrite) and m (the location of the environment variable where the malicious code is to be placed).

4 Transparent Runtime Randomization

Some custom protection mechanisms have been proposed for subclasses of UCIT vulnerabilities. Some of the vulnerabilities shown in Table 2 such as heap buffer overflow, integer overflow, and double-free as yet do not have a solution. Given that there are so many types of vulnerability, a generic mechanism is clearly preferred. We propose a general defense algorithm in this section, and we show that the alogirthm provides an effective solution to the newer heap buffer overflow, integer overflow and double-free vulnerabilities in addition to the well-studied

[4]The merged free block should be part of a doubly linked list of larger block size, which is not shown in the figure.

[5]Since `tTdvect` is an array of type `u_char`, the actual attack requires four `-D` command line options.

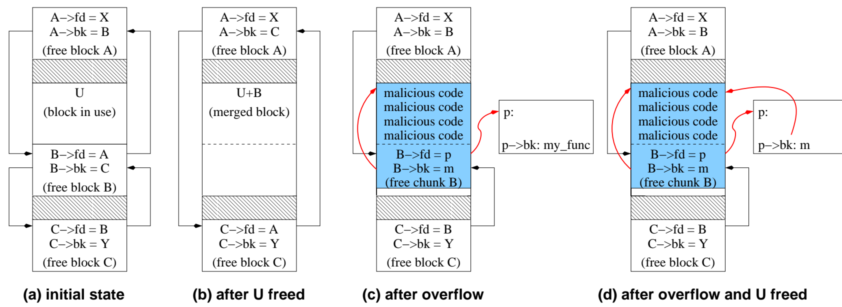


Figure 2. Malloc-Heap Overflow Attack Example

```
u_char tTdvect[100];   /* trace vector */
tTflag(char* category_flag, char* level_flag)
{
  signed int category, level;
  category = convert_to_int(category_flag);
  level    = convert_to_int(level_flag);
  if ( category >= 100 ) category = 99;
  tTdvect[ category ] = level;
}
```

Figure 3. ***sendmail*** **signed int overflow**

stack buffer overflow and format string attacks.

Recall from Table 2 that a successful attacker must correctly determine the runtime locations of memory elements, e.g., function pointers, return addresses, environment variables, command line options, library functions, and stack or heap buffers. The idea behind Transparent Runtime Randomization (TRR) is to randomize the runtime locations of these critical data elements in an application so that it is difficult (or virtually impossible) for an attacker to correctly determine their runtime locations via experimentation. In the stack-smashing attack described in Section 3, the value of m, i.e., the address (of the malicious code) that the attacker needs to determine, points to the stack buffer `msg[128]`. After TRR randomizes the runtime location of the stack, the location of `msg[128]`, i.e., the value of m, can no longer be statically determined. Hence, an attack that relies on the correct calculation of m is easily foiled.

4.1 What Can Be Randomized Transparently?

The locations of critical program elements that an attacker needs to determine for launching a successful attack (see Table 2) usually reside in well-defined memory regions in the process address space. We distinguish two types of memory regions: *position independent* and *position dependent*. A memory region is position independent if it can be freely placed in virtual memory at application startup time without breaking[6] the application, i.e., there are no inherent complex relationships with other parts of a program with respect to positioning. A memory region is position dependent if relocating it at application startup time could cause a chain of broken references from either the program code or data. A process's stack, heap, and shared libraries are position independent, while the global offset table is position dependent. The following explains the position dependency nature of these basic data regions.

- **User stack:** Before an application process begins to execute, the operating system kernel sets up the user stack and stores information such as environmental variables and command line arguments on the stack. The kernel then sets up the stack pointer (on Linux/IA-32, it is the *esp* register). The application program accesses data on the user stack through the stack pointer plus an offset. The program works correctly as long as the stack pointer is correctly initialized, hence the stack is position independent.
- **Shared libraries:** Shared libraries, also known as dynamically linked libraries, are compiled as Position Independent Code (PIC). The library functions are invoked by the program using base register plus offset and can be loaded anywhere in a program's address space as long as the base register is set up correctly.
- **User heap:** Heap is managed by dynamic memory management functions such as `malloc()` and `free()`. At runtime, `malloc()` determines the beginning of the heap via the `brk()` system call. A program accesses the heap using pointers to memory regions allocated by `malloc()`, hence the program does not make any assumptions on the runtime location of the heap.
- **Global offset table (GOT):** Once a program is compiled, the GOT is fixed at a location inside the program's static data segment. Any uncoordinated relocation of GOT will break the program, since part of the program code (the procedural linkage table or PLT) directly references GOT. As a result, GOT is position dependent, and relocation of it requires corresponding changes in the referencing code in the PLT as well.

TRR randomly relocates both position independent and position dependent regions by modifying the dynamic program loader. While relocation of position independent regions is relatively simple, relocation of position dependent regions poses a challenge to TRR. The rest of this section explains how both are implemented.

4.2 Operations of TRR

Figure 4 shows the typical sequence of steps required to launch an application, using *netscape* as an example (shadowed box in the figure represents operations inside the operating system kernel). In this example, the user types `'netscape'` at the shell prompt, and the shell creates a child process using the `fork` system call. The new child process uses the `execve` system call to load and initialize `netscape`. Inside the `execve` system call, the operating system kernel maps the executable into memory, sets up its code and static data segments, sets up its stack, heap and dynamic program loader, and then transfers control to the program loader. The dynamic program loader maps the shared libraries required by `netscape` into memory. Finally, the program loader hands over control to the entry point of `netscape`, and `netscape` begins to execute. The TRR

[6] *Breaking* a program means causing it either to crash or to output incorrect results.

operations are shown in bold italics in Figure 4 (in the shaded box). The following two sections explain the modifications the dynamic program loader.

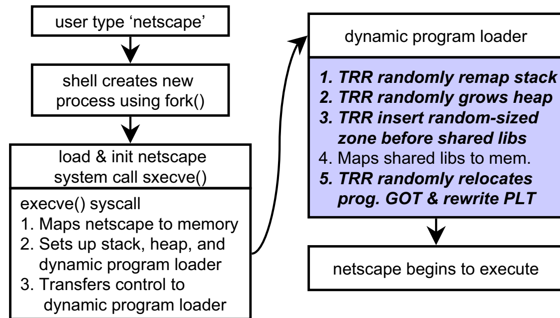


Figure 4. Operations of TRR at Process Launch

4.2.1 Relocation of Position Independent Regions

In modern Unix-based operating systems (Linux, FreeBSD, and Solaris), the memory layout of an application process is determined by the compile-time link editor, the operating system kernel, and the runtime dynamic program loader. The *compile-time link editor* determines the memory addresses of the program code, static initialized data, and static uninitialized data (BSS). Once a program is compiled, the memory location of the code and data segments are fixed. The *operating system kernel* determines the starting address for the the program heap, stack, and dynamic program loader. The *dynamic program loader* determines the memory location for shared libraries.

Figure 5 shows the memory layout for a typical application process on Linux/IA-32. Note that the address of the code segment by default begins at `0x08048000`. The static data immediately follows the text segment, and the BSS segment (uninitialized data) immediately follows the static data segment. The kernel specifies the end of data segment as the start of the heap and specifies `0x40000000` as the starting address for the dynamic program loader and shared libraries. The lower 3 gigabytes (GB) of memory address space form the user space, and the higher 1 GB is reserved for the operating system kernel. The kernel always sets up the user space stack to start at the top of the user memory address space at `0xbfffffff` (3GB - 1), and the stack grows downward. The starting addresses of heap, stack, and shared libraries are set by the `execve()` system call.

One way to randomly relocate the stack, heap, and shared

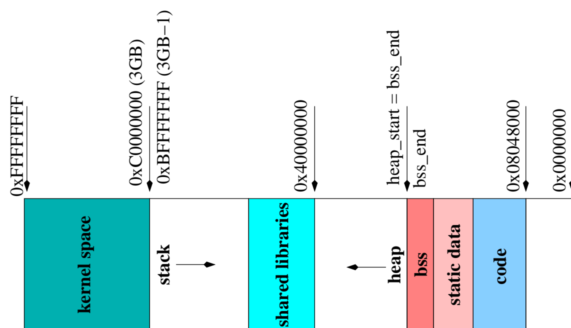


Figure 5. Memory Layout for Linux Process

libraries is to modify the `execve()` system call: when the kernel calculates and sets the base addresses for these position independent regions, a random offset is added/subtracted to the base value. This approach, however, requires modification, recompilation, and re-installation of the operating system. TRR chooses to implement the relocation algorithm in the dynamic program loader. When the operating system kernel hands over control to the dynamic program loader, the base addresses for the stack, heap, and loader have been set up. However, the loader can change these basic execution environment setups before any other code is executed, as long as the changes conform to the Application Binary Interface (ABI) standard defined for the underlying processor architecture.

The ABI for IA-32 architecture [1] specifies that the operating system kernel should place basic information about the execution environment on top of the user stack before control flow is transfered to the program loader. The information includes environment variable strings, an auxiliary information vector, the command line option strings, a vector of pointers to environment variable strings, a vector of pointers to the command line option strings, and the number of command line arguments. To randomly relocate the user stack in the program loader, TRR must preserve this information and the integrity of the pointers. This relocation is achieved through the following steps: (1) create a new stack segment at a random location below the current stack using the `mmap` system call; (2) copy the content of the old stack to the newly allocated stack, adjusting the pointers in the auxiliary, environment, and command line pointer vectors; (3) set the stack pointer register to the top of new stack; (4) free the old stack by using the `munmap` system call. After these steps, the original stack allocated by the operating system kernel no longer exists; the application program will instead use the new stack randomly relocated by TRR.

TRR randomly relocates the user heap by growing the initial heap with a random amount of space using the `brk()` system call. To randomize the location of shared libraries, TRR creates a random-sized memory mapping immediately following the dynamic program loader. When the loader maps shared libraries used by the program into the virtual memory, the mapping forces the the libraries to be loaded at a location following the random-sized region. Although random relocations for both heap and shared libraries require extra virtual memory address space, no extra physical memory space is needed. Operating systems usually allocate physical page frame only when the virtual memory page is accessed. Since the program is not aware of the added heap space and the memory mapping, no access should be made to these memory regions and hence no physical page frame should be needed.

4.2.2 Random Relocation of Position Dependent Regions

This section discusses random relocation of the position dependent element, the global offset table (GOT). The challenge in relocating a position dependent element is that an uncoordinated approach can invalidate the referencing code from the PLT and lead to a cascading broken-references. We propose an automatic

code-rewriting technique to randomly relocate the GOT while avoiding the broken reference scenario.

The GOT is a table of function pointers used for dynamic library function calls. It is located in the writable data segment of a program and has been the target of many security attacks. Once a program is compiled, the location of GOT inside the data segment of a program is fixed (see Figure 6(a)). A potential attacker write access to the GOT can change one of the function pointers in the table to point to the malicious code he/she intends to execute. To the best of our knowledge, there is as yet no good published solution to protect this table from being maliciously tampered with, other than redesigning it. Since GOT is, as defined in Section 4.1, position dependent, uncoordinated relocation can break references to it from the PLT in the code segment. To avoid broken references, the referencing instructions in the PLT are automatically rewritten and redirected to the new GOT. Before describing the relocation change, we first briefly explain how GOT and PLT interact in supporting dynamic library function calls.

GOT-PLT interaction overview. In a dynamically linked program, runtime libraries are not physically embedded; instead, only the names and version numbers of libraries are recorded in the executable. Calls to library functions are resolved at runtime. When a program executable is linked at compile time, any calls to shared libraries are directed to the PLT entries in the *read-only* code segment. PLT is essentially proxy code for shared library calls; it uses GOT in the data segment to store the addresses of shared library functions. Before the program is executed, the entries in GOT are resolved to the corresponding library functions[7]. The interaction between GOT and PLT is illustrated using a simple example in Figure 6(a). When the compiler generates code to call the library function `printf()`, it generates a call to the PLT entry for it (*plt_printf* in Figure 6(a)). The code in PLT at *plt_printf* makes an indirect jump to the real library function using the address in GOT at *got_printf* that is resolved to the library function `printf()` at initialization time.

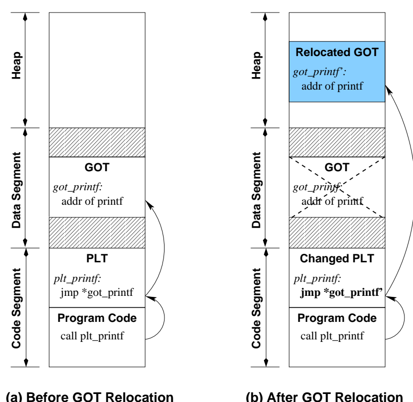


Figure 6. GOT Random Relocation

[7]The GOT entry could also be resolved lazily the first time a function is called. Our algorithm for relocation applies regardless. For presentation simplicity, we assume that it is resolved at program startup time.

To randomly relocate GOT, we also change the dynamic program loader, to allocate space for a new copy of GOT at a random memory location, and to copy the original GOT to the new location. Since the code in PLT refers to the entries in the original GOT, each entry in the PLT is rewritten to point to the corresponding entry in the new GOT. The following steps detail the procedure: (1) set the *writable* flag for the code segment; (2) go through each entry in the PLT and change the indirect jump instructions `jmp *got_entry` to `jmp *got_entry'`; (3) clear the *writable* flag for the code segment. The PLT entry *plt_printf* now has an indirect jump instruction to the new GOT entry *got_printf'* (Figure 6(b)). After the random relocation, the original GOT is no longer being used; instead, a new GOT at a dynamically determined random location is in effect, and the address of a GOT entry can no longer be statically determined. Note that since PLT is part of the *read-only* code segment, an attacker cannot write to it before seizing control of the program.

5 Effectiveness and Performance Evaluation

In this section, we evaluate the effectiveness of TRR in protecting against real-world attacks. We also present the performance overhead measurements for a set of representative applications.

5.1 Effectiveness Evaluation

The effectiveness of TRR was tested using publicly available vulnerable programs and attacks against them (from the online security bulletin board `www.securityfocus.com`). The selected applications are widely used in open systems such as Linux. The vulnerabilities and the attacks we used are presented below.

- *traceroute* is a network diagnostic tool used for finding the path on a network between two hosts. A *double-free vulnerability* exists in this program that causes it to release a memory region using `free()` while the region is not allocated through the corresponding function `malloc()`. To exploit the vulnerability, an attack uses malicious command line options to cheat `free()` into overwriting a function pointer in the program's GOT to create a local shell.
- *sendmail* is the Unix email transport agent that sends messages to remote hosts. An *integer overflow vulnerability* exists in sendmail's debugging functionality when it uses a user-supplied signed integer to address an array. As discussed in Section 3.3, the attack uses a large number to overwrite a function pointer in GOT to create a local shell.
- *ghttpd* is a fast and efficient web server capable of handling thousands of simultaneous connections. A *stack overflow vulnerability* exists in its `log()` function when a `GET` request of excessive length is sent. The attack overwrites the function return address on the stack and creates a remote shell.
- *rpc.statd* is a server that implements the Network Status and Monitor protocol as part of the Network File System

(NFS). A *format string vulnerability* exists in its call to the `syslog()` function (a `printf`-like function). The attack uses a format string that injects malicious code into the process address space and overwrites function pointers to create a remote shell.

- *null httpd* is a lightweight, multithreaded web server for Linux and Windows. A remotely exploitable *heap overflow vulnerability* exists in its handling of the `POST` request. The attack passes a negative content length using the `POST` request to trigger a heap overflow, overwrite a function pointer in GOT, and create a remote shell.

Table 3 shows the vulnerable programs and attacks used in our experiments. Without TRR, the attacks succeed in obtaining a remote or local shell. With TRR in place, the attacks cause the target vulnerable program to crash, and therefore the intruder is stopped from causing further damage to the target system. The results also demonstrate the general applicability of TRR in defeating different types of attacks, including double free, integer overflow and malloc-based heap overflow, for which no good solutions exist to date.

Program	Attack Type	No TRR	TRR
traceroute	double free	local shell	crash
sendmail	integer overflow	local shell	crash
ghttpd	stack smashing	remote shell	crash
rpc.statd	format string	remote shell	crash
null httpd	malloc-heap overflow	remote shell	crash

Table 3. Evaluation Results Against Real Attacks

5.2 Performance and Memory Overhead

The performance overhead of TRR is measured using a large set of applications that can be divided into two groups. The first group contains *network server applications* such as web, ftp, and secure shell (ssh) servers. The second group contains *stand-alone applications* such as compilers, editors, and web browsers. Since TRR is implemented at process initialization, we measure the time between the application program's entry to the `execve()` system call and the runtime system's handing over of control to the program's entry point. We measure the number of clock cycles elapsed for this period of time and convert them to microseconds using the processor clock frequency. Table 4 shows the application startup overhead averaged over 200 runs of each program. The measurement is taken on a PC with Redhat Linux 7.1 (kernel 2.4.2), Pentium IV 1.4GHz processor, and 256MB memory. The numbers in the *No TRR* column are obtained using the original Linux loader `ld-linux.so` 2.2.3. The numbers in the *TRR* column are obtained using the same versions of the loader with TRR support enabled. The overhead numbers show that TRR only introduces minor program startup overhead (2-9%). The memory overhead of TRR is essentially the size of the global offset table for a program. The *Memory Cost* column in Table 4 shows that the additional memory space required is quite small, ranging from less than 200 bytes for small applications (e.g. *traceroute*) to around 3.5 KB for very large applications (e.g., the Netscape web browser).

6 Discussion

6.1 Possible Attacks Against TRR

Proposing a new mechanism to defend against a wide range of attacks posts an important question: Is the proposed approach itself vulnerable to attacks. If so, what is the likelihood that such attacks will succeed? Below, we discuss two attacks, *brute-force* and *information disclosure*, that can potentially defeat TRR. The brute-force attack assumes a correct guess of the random offset used by TRR, and the information disclosure attack relies on possible information-leaking vulnerabilities in a program.

6.1.1 Brute-Force Attack

An attacker can try to guess the random offset used by TRR. The probability that an attacker guesses correctly is a function of the range of possible random offsets used by TRR. The larger the range, the lower the chance that it can be guessed correctly. The range of possible random offsets that can be used by TRR is limited by the following two factors: (1) the range must be small enough such that the stack, shared libraries, and the heap do not overlap (see Figure 5); (2) the range must be small enough that the above three segments have enough space to grow. The current TRR implementation uses the range between 0 and 9216 (9K). The problem is alleviated by system monitoring mechanisms such as a system administrator or an intrusion detection system: when a critical application such as a web server or mail server keeps crashing, an alarm is raised by the monitoring/detection mechanisms.

Let us now consider the probability of a successful brute-force attack. Let r be the number of values the random offset can take. Assuming uniform distribution of r, the probability that an attacker can guess correctly in a single attempt is thus $\frac{1}{r}$. Therefore, without any detection mechanism, the attacker needs $n = \frac{r}{2}$ tries, on average, to break into a system. With an intrusion detection system in place, what is the probability that an attacker can guess correctly before it is detected? Let d be the number of crashes the detection mechanism can tolerate before it raises an alarm. The probability p, that the attacker can guess right before being detected is then:

$$p = \sum_{i=1}^{d} \frac{1}{r} \cdot (1 - \frac{1}{r})^{i-1} = \frac{1}{r} \cdot \sum_{i=1}^{d} (1 - \frac{1}{r})^{i-1}$$

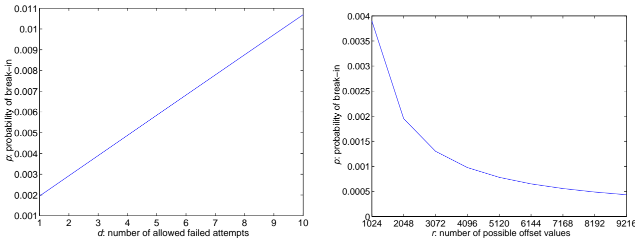
Therefore, the expected number of sites (n), the attacker needs to attack before breaking into one of them can be expressed as:

$$n = 1/p = \frac{r}{\sum_{i=1}^{d} (1 - \frac{1}{r})^{i-1}}$$

Figure 7 plots the probability of break-in when varying the intrusion tolerance level is d and the number of possible random offset values r. The figures show that with TRR protection in place, the probability of break-in is remote.

Program	Description	Startup Time (μsec)			Memory Cost (bytes)
		No TRR	TRR	Overhead	
apache	web server	1.94	2.01	3.82%	816
ghttpd	web server	0.96	1.02	6.96%	248
wu-ftpd	FTP server	1.71	1.84	7.87%	724
lpd	printer server	1.72	1.79	4.31%	736
nullhttpd	web server	1.23	1.30	5.49%	296
opensshd	secure shell server	2.72	2.87	5.34%	1088
rpc.statd	NFS stat server	0.96	1.04	8.71%	304
XFree86	X window server	1.93	2.07	6.97%	928
cc1	GNU C compiler	0.96	1.02	6.06%	324
gvim	text editor	16.57	17.28	4.26%	2988
netscape	web browser	8.39	8.93	6.44%	3668
pine	email client	6.69	7.13	6.58%	992
traceroute	network diagnosis tool	0.96	1.02	5.83%	196
xemacs-21.1.14	text editor	22.73	23.31	2.54%	2088

Table 4. Application Start-up Overhead and Additional Memory Cost



(a) varying d (no. of tolerated failures) for $r = 1024$

(b) varying r (no. of random values) for $d = 3$

Figure 7. Probability of Brute-force Attack

6.1.2 Information Disclosure Attack

Additional vulnerabilities can cause a program to leak runtime memory layout information. For example, information on a program's stack such as return addresses and frame pointers can be used to derive the initial starting address of a stack that has been randomized by TRR. Information disclosure like this can be used by an attacker to defeat the TRR mechanism. When a program that requires multiple rounds of interaction between the client and the server has such a vulnerability, the attacker can initiate many rounds of conversations with the server program and form a reliable evaluation of the memory layout before launching the attack. The *FTP SITE EXEC* bug [7] is such a vulnerability for the specific version of wu-ftp 2.6.0. In this case, the remote FTP client can send a malicious input string to force the vulnerable FTP server to output contents of its runtime stack and therefore leak memory layout information. In practice, however, this kind of case is rare. In more than one hundred advisories issued by CERT over the past four years, this is the only case we found that can lead to information disclosure. In most cases, a program does not leak information, and the attacker has just one try at the target program to seize control. While the threat exists, it is not too serious for most programs.

6.2 Runtime Rerandomization

The TRR mechanism is applied to programs at process loading time. For long-running programs, for example server applications, once the process is started, the random memory layout will remain fixed until the program terminates and is reloaded. From a security perspective, a large window of static behavior can lead to vulnerability. In this section, we discuss how TRR can be dynamically applied to applications to re-randomize memory layout. The difficulty in rearranging the memory layout of a running process is that an application process's internal state may be dependent on its memory layout; for example, it may hold pointer references to its heap, stack, or shared libraries. Hence, the online memory re-randomization algorithm must be application specific. We propose two possible solutions for a class of server applications that are considered to be stateless. We illustrate the idea in the context of the Apache web server.

The Apache web server version 1.3.x has a parent process, p, that monitors a pool of servicing processes $s_1 \dots s_n$ that fulfill HTTP requests. One way to re-randomize online is to use Apache's built-in shutdown facility to gracefully stop the current invocation and reload the entire set of Apache processes. A re-randomization manager, m, can periodically shutdown and restart Apache. The advantage of this approach is that no change is required to Apache itself. The disadvantage is that graceful shutdown takes time, since p needs to wait for all pending HTTP requests to be fulfilled before $s_1, \dots, s_n$ can be stopped, which might incur undesirable downtime. A second approach, which minimizes the downtime due to reloading, requires minor changes to the Apache source code. In this approach, we can change the implementation of p as follows: periodically, p stops accepting connections and creates a new monitor process p'. The new process, p', reloads itself, creates a new pool of servicing processes, $s'_1 \dots s'_n$, and starts accepting connections immediately. Meanwhile, the old process p waits until $s_1, \dots, s_n$ complete their pending requests before shutting down. The sec-

ond approach minimizes disruption to service, since the new server can service requests before the old one shuts down.

6.3 Portability

On modern Unix-like platforms, program executables use the standard ELF (Executable and Linkable Format) format. The ELF file format determines the runtime memory layout for the application. Memory layouts on different Unix-based platforms all look like the image shown in Figure 5, although the default values for different sections might vary. Furthermore, the mechanism used for dynamic library function calls are also specified by the ELF file format. Since TRR is implemented by changing the ELF-related components, it can be ported to other Unix-like systems. On the Windows platform, executables use PE (portable executable) file format, which is different from ELF. We are currently investigating how TRR can be applied on Windows.

7 Conclusions

This paper proposes Transparent Runtime Randomization (TRR), a generalized approach to protecting systems against a wide range of security attacks that exploit vulnerabilities leading to to unauthorized control information tampering (UCIT). The undelying principle of TRR is to randomize the application memory layout so that it is virtually impossible to determine locations of critical program data such as buffers, return addresses, and function pointers. The randomization algorithm is fully transparent to application programs because it is implemented by modifying the dynamic program loader. Because TRR is applied dynamically at runtime rather than statically at compile time, each invocation of a program has a different memory layout. The effectiveness of TRR has been tested using real-world security attacks. Results show that TRR can defeat a wide range of attacks including malloc-based heap overflow, integer overflow, and double free, for which effective protection mechanisms are yet to emerge. Performance measurements show that TRR incurs only a small overhead (2-9%) when launching applications and no overhead is incurred once the application begins execution. TRR is also portable to other Unix-like platforms.

8 Acknowledgement

This work is supported in part by a grant from Motorola Inc. as part of Motorola Center for Communications, and in part by MURI Grant N00014-01-1-0576. We thank Fran Baker for her careful proof-reading of the manuscript.

References

[1] *System V Application Binary Interface - Intel386(TM) Arhcitecture Processor Supplement*. 4th edition, 1997.

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

[3] Anonymous. Once upon a free(). *Phrack Magazine*, 57(9), Aug. 2001.

[4] A. A. Avizienis. The methodology of n-version programming. In M. R. Lyu, editor, *Software Fault Tolerance*, chapter 2, pages 23–46. John Wiley and Sons, 1995.

[5] C. Bain, D. Faatz, A. Fayad, and D. Williams. Diversity as a defense strategy in information. In *MITRE Corporation Technical Paper*, 2001.

[6] A. Baratloo, T. Tsai, and N. Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings. of the USENIX Annual Technical Conference*, June 2000.

[7] CERT. Cert advisory ca-2000-13 two input validation problems in FTPD. *http://www.cert.org/advisories/CA-2000-13.html*, 2000.

[8] S. Chen, J. Zymla, and Z. Yang. N-version programming for stack overflow detection. *ECE442 Design of Reliable Systems and Networks Class Project Report, University of Illinois at Urbana-Champaign*, Dec. 2000.

[9] C. Cowan, M. Barringer, S. Beattie, , and G. Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Conference*, 2001.

[10] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*, 1998.

[11] Y. Deswarte, K. Kanoun, and J.-C. Laprie. Diversity against accidental and deliberate faults. In *Computer Security, Dependability, and Assurance: From Needs to Solutions*, Nov. 1998.

[12] D. Evans. Static detection of dynamic memory errors. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[13] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[14] S. C. Johnson. Lint, a C Program Checker. *Bell Laboratories Computer Science Technical Report 65*, Dec. 1977.

[15] M. M. Kaempf. Vudo - An object supersitiously believed to embody magical powers. *Phrack Magazine*, 57(12), Aug. 2001.

[16] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. 10th USENIX Security Symposium*, Aug. 2001.

[17] NetCraft. Netcraft web server survey. *http://www.netcraft.com/survey/*, Oct. 2002.

[18] OpenWall Project. Linux kernel patch from the Openwall project. *http://www.openwall.com/linux/*.

[19] PaX Team. Homepage of The PaX Team. *http://pageexec.virtualave.net/*.

[20] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, Copper Mountain, Colorado, 1995.

[21] C. Pu, A. Black, C. Cowan, J. Walpole, and C. Consel. A specialization toolkit to increase the diversity of operating systems. In *ICMAS Workshop on Immunity-Based Systems*, 1996.

[22] Sendmail Consortium. Sendmail home page. *http://www.sendmail.org/*.

[23] Unix System Labs. Executable and Linkable Format (ELF) Version 1.1. *Tool Interface Standards*.

[24] Vendicator. Stack Shield. *http://www.angelfire.com/sk/stackshield/*, 2000.

[25] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of 7th Network and Distributed System Security Symposium*, Feb. 2000.

[26] R. Wojtczuk. Defeating Solar Designer non-executable stack patch. *http://www.insecure.org/sploits/non-executable.stack.problems.html*, Jan. 1998.