RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization

Xuxian Jiang[†], Helen J. Wang[‡], Dongyan Xu[∗] Yi-Min Wang[‡]

[†] George Mason University	[‡] Microsoft Research	[∗] Purdue University
xjiang@ise.gmu.edu	{helenw, ymwang}@microsoft.com	dxu@cs.purdue.edu

Abstract

Code injection attacks are a top threat to today's Internet. With zero-day attacks on the rise, randomization techniques have been introduced to diversify software and operation systems of networked hosts so that attacks that succeed on one host cannot succeed on others. Two most notable system-wide randomization techniques are Instruction Set Randomization (ISR) and Address Space Layout Randomization (ASLR). The former randomizes instruction set for each process, while the latter randomizes the memory address space layout. Both suffer from a number of attacks. In this paper, we advocate and demonstrate that by combining ISR and ASLR effectively, we can offer much more robust protection than each of them individually. However, trivial combination of both schemes is not *sufficient. To this end, we make the key observation that system call instructions matter the most to attackers for code injection. Our system,* RandSys*, uses system call instruction randomization and the general technique of ASLR along with a number of new enhancements to thwart code injection attacks. We have built a prototype for both Linux and Windows platforms. Our experiments show that RandSys can effectively thwart a wide variety of code injection attacks with a small overhead.*

1 Introduction

A prevalent form of attacks on the Internet, commonly known as *code injection attacks*, is to exploit a software vulnerability on a host and cause malicious execution of either injected attack code or pre-existing code (such as *libc* functions). Such attacks can exploit many vulnerability types, such as input validation errors, exception condition errors, and race conditions. Code injection attacks pose serious threat to the Internet: fast- and wide-spreading worms such as CodeRed [3], Blaster [5], and Sasser [6] all depend on the successful execution of injected code to complete their infections and replications. In this paper, we focus on *remote machine-code injection attacks*, but *not* on other injection attacks, such as SQL injection and Cross-Site Scripting attacks. For the purpose of exposition, we use the conventional term "shellcode" to refer to the injected code.

While patches can protect known vulnerabilities, zero day exploits are on the rise [7] and demand a more proactive approach. Forrest et al [22] advocated building diversity into software and operating systems of networked hosts in the first place. There are two main system-wide randomization techniques proposed since: Instruction Set Randomization (ISR) [12, 13, 24, 33] and Address Space Layout Randomization (ASLR) [2, 14, 15, 37]. ISR creates a randomized instruction set for each process so that instructions in shellcode fail to execute correctly even though attackers have already hijacked the control flow of the vulnerable process. ASLR, instead, randomizes the memory address layout of a running process (including library, heap, stack, and relative distances between data and code) [2, 14, 15, 37] so that it is hard for attackers to locate injected shellcode or existing program code, preventing attackers from hijacking the control flow.

Both randomization schemes suffer from a number of attacks. ISR is vulnerable to attacks that avoid using injected machine instructions. For example, ISR suffers from *return-into-libc* attacks [12, 13, 24] in which attackers call pre-existing library functions (e.g., $system()$) without the need of injecting malicious instructions. Meanwhile, ASLR suffers from attacks that avoid using *specific* memory addresses. Although ASLR makes control-flow hijacking more difficult, shellcode locations might still be easy to guess. For example, a new form of attack which we call "*code spraying*" attacks, could exploit a buggy application behavior and "spray" a shellcode repetitively throughout *large* write-able user-level memory areas (say 256MB) — this leaves only 4 bit entropy in the current 32 bit architecture for attackers to guess the location of a shellcode replica. Furthermore, control data can be overwritten without knowing their precise location. For example, attackers can overflow a memory area that likely contains a code pointer, with repetitive guessed addresses [14] – an attack termed as "*address spraying*".

In this paper, we advocate and demonstrate that by combining ISR and ASLR *effectively*, we can offer much more robust protection than each of them individually. Although a trivial combination of ISR and ASLR can address the aforementioned attacks, such a system *cannot* be prac-

tically deployed. The reason is that ISR incurs prohibitive performance overhead because of its per-instruction de-randomization and lack of hardware support [12, 13, 24]. Here, we make the key observation that system call instructions are almost always used by shellcode to carry out its malicious actions. Therefore, we can simply randomize the system call instructions which matter the most to attackers and significantly reduce the ISR overhead. Our system, called *RandSys*, uses system call instruction randomization and the general technique of ASLR to thwart code injection attacks. We refer to system call instructions and their associate library APIs as *system service interface*. RandSys performs randomization at the process load time by instrumenting the process with a thin transparent virtualization layer that randomizes system service interface; while at run-time, it de-randomizes the instrumented interface for correct execution.

However, by randomizing only selective instructions, attackers have more power if they can overcome the ASLR part of the scheme and hijack the control flow. To this end, we strengthen the state-of-the-art ASLR schemes with a number of new techniques. We perform function name randomization so that function import and export tables are essentially encrypted and attackers are unable to hand-craft assembly code to access these tables. Furthermore, we employ "decoys" in the function export table pointing to access-protected "guard pages", so that RandSys can undermine "function fingerprinting" attacks that walk through function export tables and look for a known function fingerprint. We also carefully manage randomization in function import and export tables so that attackers cannot correlate two tables in finding a function.

RandSys raises the bar for code injection attacks significantly. To launch a successful attack, attackers would need to mount kernel code injection attacks or *non-control-data* attacks [16]. RandSys does *not* defeat kernel code injection attacks because it targets user-level attacks by randomizing system service interface between user programs and the kernel. In non-control-data attacks, security-critical application data (such as configurations, user input, or decision-making data) rather than control data (such as return addresses or function pointers) are corrupted by memory error exploits. In such attacks, since code injection may not happen in the first place, RandSys would not be effective. In addition, RandSys may cause disruptions to programs with self-modifying code, where a system service invocation instruction may be dynamically created. Another limitation of RandSys is that it makes debugging and diagnostic tasks more difficult, a common problem in randomization-based techniques.

We have built a prototype of RandSys in both Linux and Windows platforms. Our experiments show that RandSys can defeat a wide variety of code injection attacks while incurring low performance penalty. RandSys is independent of vulnerability-specific details, and hence can defeat zero-day attacks. Our RandSys prototype has successfully thwarted attacks on the Windows JView Profiler vulnerability (MS05-037/July, 2005) and the Microsoft Visual Studio .NET "msdds.dll" vulnerability (August 17, 2005) *before* their patches became available. RandSys readily supports all the applications in our experiments, including the Apache/IIS web server, various FTP daemons, Internet Explorer and Firefox web browsers.

In the rest of the paper, we first present the RandSys design in Section 2. We then give a detailed security analysis of RandSys in Section 3. We describe the RandSys implementation in Section 4 and demonstrate its effectiveness against a number of real-world attacks in Section 5. We compare RandSys with related work in Section 6. Finally, we conclude in Section 7.

2 RandSys Design

In this section, we first present our design of load-time randomization and run-time de-randomization schemes in RandSys. Then we present a method for dynamic code injection detection as our next line of defense.

2.1 Load-Time Randomization

System Call Load-Time Randomization When a process is created, RandSys takes over the control (e.g., intercepting the *sys_execve* system call in the kernel) before program execution. RandSys searches for system call invocations, such as "int $0x80" in Linux and "int $0x2e" or "sysenter" in Windows. For each identified system call i at memory location L_i, the original system call number S_i^o is overwritten with a new, randomized system call number S_i^n using the following equation:

$$S_i^n = R_K(S_i^o, L_i).$$

R_K is our load-time system-call randomization algorithm using key K. R_K takes two parameters: the original system call number S_i^o, and the location of the call L_i. Note that even the same system call at different locations will yield different call numbers. We maintain the key K in the kernel space. And we used DES encryption in our prototype. A more aggressive scheme can further randomize the system-call calling convention, such as permuting the roles among EAX, EBX, ECX, and EDX registers or padding system call parameters.

In Windows, dynamically linked libraries may be loaded into a process at run-time. In RandSys, we instrument and randomize system calls in these libraries by intercepting library-loading APIs (e.g., "LoadLibraryA"). Note that an attacker may attempt to misuse this support. We defer the related security analysis to Section 3.

Library API Load-Time Randomization RandSys enables two types of library API randomization: library re-mapping and function randomization.

Library re-mapping is an existing ASLR technique, which renders exploits (e.g., regular *return-into-libc* attacks) that depend on predetermined memory addresses

useless. Library re-mapping randomizes library base addresses and re-organizes internal functions. Randomizing the library base addresses makes it hard to predict the *absolute address* of a library. Re-organizing internal functions makes the *relative address*-based attacks unlikely to succeed. The re-mapping modifies the import and export function tables used by dynamic linking. For example, reorganizing exported functions alters the *.dynamic/.dynstr* section in Linux or the Export Address Table (EAT) in Windows, while re-organizing imported functions modifies the PLT/GOT component in Linux and the Import Address Table (IAT) in Windows. Library re-mapping does not need to be de-randomized at run-time since function import and export tables already contain randomized function locations.

Function randomization is one of our new enhancements to strengthen existing ASLR schemes. It provides function name randomization and API calling convention shuffling. Function name randomization makes function name-lookup unique to each process, while API calling convention shuffling randomizes the *run-time* API interface by shuffling existing parameters and padding new ones. Function randomization is needed because we want to prevent attackers from handcrafting machine code to access function import and export tables and to look for the randomized location of desired function names.

Name randomization replaces a function name with another randomized name string. We note that a naive name randomization scheme that generates an identical function name for both the import library and the export library would suffer from the *correlation attack*. An attacker can correlate the imported function names from one library (e.g., through IAT in Windows) with the exported function names in another (e.g., through EAT), and infers the function. To counter this attack, name randomization applies different randomization algorithms based on whether the function is imported or exported: (1) If a function is exported to other library modules, the corresponding function name F_E^o is randomized to another name string $F_E^n = R_E(F_E^o)$, where R_E is the randomization algorithm applied to the exported function names. (2) If a function is imported by module M_i, the imported function name F_I^o is randomized to another name $F_I^n = R_I(F_I^o, M_i)$, where R_I is the randomization algorithm with two parameters: the imported function names and the run-time base address of the importing library module M_i. Note that although different modules may import the same function, R_I generates different randomized names. (3) Finally, the name inconsistency caused by these two different randomization functions can be resolved at run-time by a dedicated process-specific name resolution routine, such as a customized *dl_runtime_resolve()* in Linux or *GetProcAddress()* in Windows.

Function fingerprinting is a commonly used attack technique. One variant of such technique scans the function export tables and searches for a known function fingerprint that is in the form of either an instruction sequence or the function's hash value. To combat this type of attack, we add "decoy" entries to the function import and export tables; each decoy entry points to a guard page, which is a page with the access protection such as *PROT_NONE* in Linux or *PAGE_NOACCESS* in Windows. Any attempt to read, write, or execute on a guard page will result in an access violation exception.

2.2 Run-Time De-randomization

System Call De-randomization The execution of the system call instruction (e.g., "int $0x80" in Linux or "int $0x2e" or "sysenter" in Windows) generates a software trap to kernel mode and invokes the system call dispatcher. The system call dispatcher dispatches the system service routine according to the register that contains the system call number (e.g., EAX). In RandSys, we customize the system call dispatcher to perform de-randomization. The dispatcher first inspects the stack or its context environment to derive the actual memory location L_i at which a system call i with randomized system call number S_i^n is made. RandSys then recovers the original system call number $S_i^o = R_K^{-1}(S_i^n, L_i)$ where R_K^{-1} is the run-time de-randomization algorithm of its load-time counterpart R_K.

Function Name Resolution As described in Section 2.1, function name randomization purposely causes name inconsistency between functions in export table and the same functions imported by other modules in their respective import tables. To resolve this inconsistency, we use a run-time name resolution function R_R which maps a randomized imported function name to its corresponding randomized exported function name with the import module base address M_i as a parameter: $R_R(R_I(plaintext\_function\_name, M_i), M_i) = R_E(plaintext\_function\_name)$.

2.3 Dynamic Injection Detection

One attack against RandSys is to identify and jump to existing application code (including *libc* functions) that invokes system service interface. To this end, we develop a dynamic injection detection scheme to enable *defensive* execution of the existing program code, including the detection and termination of a shellcode execution. Since a shellcode is dynamically injected into a running process, the code page containing the shellcode needs to be writable for the injection. However, at the same time, the shellcode is not a part of the original program code. Hence, there are two inherent characteristics associated with the code page containing the shellcode: (1) it is *writable*; and (2) it is *not* mapped from the executable file. Note that these two characteristics will not be exhibited in any normal program that does not contain any self-modifying code. Based on this observation, we use the following heuristics to detect shellcode's existence on

a page when an existing system call or library function is invoked:

```
DYNAMICINJECTIONDETECTION(EBP)
 1   depth ← 0
 2   while ISSTACKFRAMEVALID(EBP)
 3       and (depth ≤ BACKTRACE_DEPTH)
 4       do return_addr ← GETRETURNADDR(EBP)
 5          code_page ← GETPAGEFROMADDR(return_addr)
 6          if ISPAGEWRITABLE(code_page)or not
 7              DOESPAGECOMEFROMFILE(code_page)
 8              then return INJECTION_DETECTED
 9          EBP ← GETNEXTFRAME(EBP); depth ← depth + 1
10   return UNDETECTED
```

Essentially, the detection algorithm is a recursive stack-based inspection algorithm, which traverses the stack frame to assess whether the code page containing the return address matches these two characteristics. Dynamic injection detection can be performed for any library API (within its prologue or epilogue). In addition, the system call dispatcher, which performs run-time system call de-randomization, can also be extended to perform this task.

3   Security Analysis

Attacks Using Direct System Service Invocation An attacker may directly use system calls in shellcode. The system call randomization of RandSys easily defeats such straightforward attacks. Furthermore, RandSys is resilient to replay attacks where attackers re-use randomized system calls. This is because our randomization algorithm takes the memory location of a system call as a parameter — two system calls with the same system call number will be de-randomized into two *different* system call numbers since they are at different locations.

Attackers may attempt to acquire the randomization key directly. This attempt is also defeated by RandSys. The reason is that the randomization key is stored in the kernel space; and user-level programs are unable to get the randomization key. Note RandSys is not effective against kernel-level code injection attacks which could be used to tamper the key or carry out other malicious actions.

Attackers could also try to construct plaintext-ciphertext pairs to brute-force the key. RandSys makes this very difficult. Firstly, a strong encryption algorithm and a long key makes it almost impossible to crack the key. Secondly, because our randomization algorithm is location-dependent, attackers are forced to scan code memory to collect the precise locations as well as the semantics of the instructions. Our decoy and guard page mechanisms (Section 2.1) can detect and undermine such scanning activity. Lastly, our dynamic injection detection technique in Section 2.3 serves as another line of defense.

Attacks Using Indirect System Service Invocation Instead of invoking system service interface directly, an attacker may try to reuse existing system service invocations in the vulnerable program. To this end, an attacker must first accurately locate the memory location of the desired system call or associated library API invocation instructions, and then branch to that location to eventually invoke the intended system service. RandSys makes such attacks hard to succeed in a number of ways:

Firstly, the use of ASLR makes the memory location of both shellcode and pre-existing code (e.g., libc functions) hard to predict, and hence effectively defeats *return-into-libc* attacks and making control flow hijacking difficult. An advanced form of the *return-into-libc* attack, called *return-into-dl* attack was introduced by Nergal to compromise PaX [2] – a representative ASLR implementation [29]. In this attack, attackers do not directly invoke a *libc* function. Instead, it "returns" to the dynamic linker's functions (e.g., *dl_runtime_resolve()*) to look up the randomized location of the desired function by its name. RandSys can defeat this attack in two ways: (1) The *dl_runtime_resolve* function (or *GetProcAddress* in Windows) is randomized by library-remapping; (2) Even if the attacker can handcraft *dl_runtime_resolve* function (or *GetProcAddress* in Windows) to directly access function import or export tables for randomized function locations, our function randomization mechanism (Section 2.1) effectively undermines such attempts.

Secondly, even if attackers can successfully hijack the control flow of a process, since RandSys randomizes system calls and their associated library APIs, the only way for attackers to invoke system services is to find the memory locations of the desired system service-invocation instructions in the pre-existing program code. Such memory-scanning activity can be efficiently undermined by our trap mechanisms such as decoys and guard pages. Although it is possible for attack code to peek through the stack, find the location of a particular function, and then calculate the offset of the intended system service call within the function, such approach requires an in-depth understanding of run-time program stacks (and possibly program semantics).

Lastly, even if the memory location of desired system service invocation in the pre-existing program code is identified, the attack code still faces the challenge of regaining control after unidirectionally reaching that location. The reason is that a remote attack often needs to chain together a sequence of system service calls to achieve its goal.

We point out that Nergal et al [29] introduced two main techniques, "*esp-lifting*" and "*frame-faking*", for chaining system service invocations. These techniques manipulate the stack, such as lifting the ESP register or forging a stack frame, to regain the control after one *libc* call is invoked. However, both approaches have their own limitations: as acknowledged in [29], "esp-lifting" is only applicable for those binaries compiled with a certain optimization switch, i.e., *-fomit-frame-pointer*; and "frame-faking" must be aware of the precise locations of those fake frames — this can be effectively defeated by RandSys. Further-
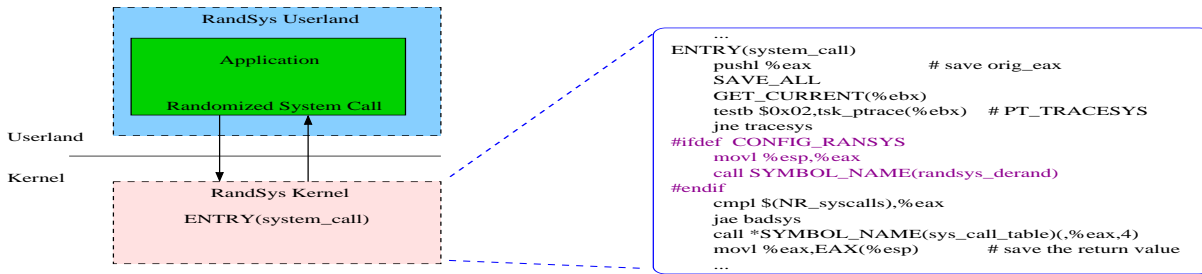
Figure 1. System Call Randomization and De-randomization in RandSys (Linux Version)

more, both techniques can be mitigated by RandSys' dynamic injection detection since the detection algorithm in Section 2.3 can be simply extended to detect the existence of those "esp lifting" or "frame faking" instructions.

Now, we examine another threat: as RandSys supports run-time library loading (Section 2.1), attackers might attempt to *abuse* this support to make an illegitimate library loading. More specifically, after circumventing ASLR and hijacking the control flow, an attacker may intentionally invoke *LoadLibraryA* to load a library with intended functions. Since this library call needs to make several system calls (e.g., reading files from the disk) and RandSys thwarts illegitimate direct system calls and captures illegitimate direct invocation of the pre-existing *LoadLibraryA* code, attackers must rely on pre-existing program code to indirectly invoke the *LoadLibraryA* call and then come up with a way to re-capture control after loading the library. Based on our earlier discussion, RandSys make such attempts hard to succeed.

4 Implementation

In this section, we describe the RandSys proof-of-concept implementation in both Linux and Windows platforms. Due to space constraint, system call randomization will be mainly described in the context of Linux platforms while library API randomization will be presented by focusing on Windows platforms.

4.1 Execution Control Interception

Load-time control interception In Linux-based systems, RandSys intercepts the *sys_execve* system call and then applies load time randomization (Section 2.1). For Windows, the implementation is different: A DLL library is first injected to existing running processes and the DLL library will hook a number of critical library APIs, including *CreateProcess()*. Once a new process is created, the hooked *CreateProcess()* will create the new process in a suspended state and then perform the necessary load time randomization before resuming process execution.

Run-time control interception Run-time control interception mainly involves the system call de-randomization and library API name resolution. RandSys has a kernel module to patch the system call dispatcher

so that it can transparently convert a randomized system call number to its original number. To achieve transparent library API name resolution, RandSys hooks a number of related function calls such as *dlsym()* in Linux and *Get-ProcAddress()* in Windows. To support run-time library loading, additional functions such as *LoadLibraryA()* in Windows are also modified.

Exception interception The introduction of decoy entries and guard pages (Section 2.1) provides an opportunity to detect and identify illegitimate read or execute accesses. RandSys hooks the exception handler, i.e., SIGSEGV in Linux and the Structured Exception Handler (SEH) [8] in Windows. More specifically, our Windows prototype hooks the *KiUserExceptionDispatcher* API, which is exported by *ntdll.dll*, to intercept the exception raised by the process. Once an exception is intercepted, RandSys checks whether it is caused by reference to a decoy entry. If not, the exception will be passed to the normal SEH chain. Otherwise, it is considered as an illegitimate access and the current prototype will attempt to terminate the mis-behaving process.

We would like to point out that exception interception can be leveraged to thwart *brute-force attacks*. Existing works [32, 34] have demonstrated that the brute-force attack is able to defeat both ISR [12, 24] and ASLR [2] schemes. However, the detection of brute-force attacks is relatively easy because they will result in frequent crashes in the victim processes. Since our RandSys prototype directly intercepts possible exceptions before they are dispatched, it is by design robust against brute-force attacks.

4.2 System Calls Randomization

After gaining the execution control at load-time, RandSys will first attempt to locate those instructions making system calls. It can disassemble all process code segments and find the system call instructions, i.e., "int $0x80" in Linux or "int $0x2e/sysenter" in Windows. However, this may incur considerable load-time latency. An alternative is to perform an offline analysis to identify the system call locations (Section 5.1). For each system call occurrence, the original system call number will be randomized (Section 2.1) as another system call number, which can later be interpreted by the RandSys ker-

	Red Hat Linux 8.0		Windows XP Professional (SP2)				
	libc-2.2.93.so	*ld-2.2.93.so*	ntdll.dll	user32.dll	gdi32.dll	imm32.dll	winsrv.dll
# System Calls	235	39	284	266	366	18	21

Table 1. Sample Library Modules and Number of System Calls in Each Module

nel. Once the new system call number is calculated, the instruction assigning the original system call number to the EAX register will be instrumented to reflect the new system call number. Figure 1 shows how the original Linux system call dispatcher, i.e., *ENTRY(system_call)*, is modified to support RandSys. Note that the RandSys kernel *SYMBOL_NAME(randsys_derand)* needs to inspect the stack to locate the exact calling location, which is needed to recover the original system call number. Table 1 shows a number of library modules and the number of system calls within each library module.

4.3 Library API Randomization

Library re-mapping Right after a new process is created but before its instructions are executed, RandSys will take over its execution, inspect the loaded modules, and attempt to re-map or re-base these modules to other random locations. As mentioned in Section 2.1, library re-mapping requires certain modifications to IAT/EAT table entries affected. The purpose of re-mapping libraries is to make their absolute and relative addresses less predictable. In addition, special decoy entries are intentionally planted to trap possible illegitimate references.

Function randomization RandSys intercepts two important function calls, i.e., *LoadLibraryA()* and *GetProcAddress()*. The first function is extensively used by Windows systems to enable run-time library loading and needs to be intercepted to perform delayed load-time randomization. The second function is also extensively used by Windows systems to resolve a function based on its string name. Since the function names will be randomized differently based on their resident modules, the interception of *GetProcAddress()* is necessary to resolve possible name inconsistency. Note both Windows and Linux have a well-defined interface to resolve functions at run-time, making this randomization procedure straightforward.

5 Evaluation

In this section, we first present RandSys latency measurement results in Section 5.1. We then present a number of experiments with more than 60 real code injection attacks, including those attacks from well-known self-propagating worms (Section 5.2). As RandSys does not require any prior knowledge about vulnerabilities and their exploitation means, RandSys is effective against zero-day exploits. This capability is demonstrated by results from two zero-day "in-the-wild" exploits, which did not have any software patch when we conducted our experiments (Section 5.3).

5.1 RandSys Latency

By performing load-time randomization and run-time de-randomization, RandSys introduces both load-time and run-time latency to the protected process. To measure the latency, we set up two physical hosts (with alias RANSYS_LINUX and RANSYS_WIN, respectively). RANSYS_LINUX is a Dell desktop PC running Red Hat Linux 8.0 with 596.913MHz Intel Pentium III (Katmai) processor and 384MB RAM while RANSYS_WIN is another Dell desktop PC running Windows XP Professional (SP2) with 2.2GHZ Intel Xeon processor and 512MB RAM. We use several popular applications for RandSys latency measurement. The results are shown in Table 2.

Table 2 indicates that RandSys with online disassembly incurs much longer load-time latency than RandSys using offline analysis. It may appear that the load time due to online disassembly is unacceptable to frequently used applications. However, we note that the disassembly only needs to be performed *once* when a new application is first introduced. The disassembly result can be reused in future runs without incurring the disassembly latency again. Table 2 also shows that system call de-randomization only introduces a small performance degradation, which is largely caused by the de-randomization algorithm. The DES algorithm usually takes only $1,200$ CPU cycles (2 microseconds) to perform decryption.

5.2 Thwarting Existing Code-Injection Attacks

We have experimented with over 60 existing code-injection attacks. RandSys is able to thwart all these attacks. Table 3 shows a selected subset of those attacks, including the recent Zotob worm [9]. Especially, the last column of Table 3 highlights the thwarting techniques from RandSys that defeat the corresponding attacks. In the following, we choose four representative attacks by the Lion worm [4], Slapper worm [30], MSBlast worm [5], and Sasser worm [6] to elaborate how RandSys successfully corrupts their infections. Due to space constraint, an in-depth analysis of their attack code is omitted in this paper and interested readers are referred to [10].

Effectiveness of system call randomization It is interesting to point out that the two shellcodes injected by the Lion worm and the Slapper worm have very similar functionality: when the shellcode in either Lion or Slapper worms is executed, it first searches for the socket of the active TCP connection established with the attacking machine and reuses this connection for further infection such as spawning a shell. More specifically, the shellcode cycles through all the file descriptors and issues a

	Red Hat Linux 8.0		Windows XP Professional (SP2)	
	Apache Web Server *(httpd-2.0.40-8)*	vsftpd FTP Server *(vsftpd-1.1.0-1)*	Internet Explorer 6.0	IIS Server 6.0
Load-Time Latency (Online Disassembly)	11.1 (seconds)	3.9 (seconds)	> 1 (minute)	> 1 (minute)
Load-Time Latency (Offline Analysis)	0.3 seconds	0.3 seconds	0.5 seconds	0.5 seconds
Run-Time Latency	1500 cycles/syscall	1500 cycles/syscall	1650 cycles/syscall	1650 cycles/syscall

Table 2. Load-time and Run-time Latency of RandSys

Attack	Reference	Description	Platform	Thwarting RandSys Techniques
CodeRed	MS01-033 CAN-2001-0500	Unchecked Buffer in the Index Server ISAPI Extension	Windows	Enhanced ASLR (EAT Randomization)
Slammer	MS02-039 CAN-2002-0649	Buffer Overrun in the SQL Server 2000 Resolution Service	Windows	Enhanced ASLR (IAT Randomization)
MSBlast	MS03-026 CAN-2003-0352	Buffer Overrun in the RPC DCOM service	Windows	Enhanced ASLR (EAT Randomization)
Sasser	MS04-011 CAN-2003-0533	Buffer Overrun in the LSASS service	Windows	Enhanced ASLR (EAT Randomization)
Witty	CAN-2004-0362	ICQ Parsing Vul. in the ISS Protocol Analysis Module (PAM) component	Windows	Enhanced ASLR (EAT Randomization)
Zotob	MS05-039 CAN-2005-1983	Buffer Overrun in the Plug and Play service (August 14, 2005)	Windows	Enhanced ASLR (EAT Randomization)
Ramen	CVE-2000-0917 CVE-2000-0573 CVE-2000-0666	LPRng Format String Bug WU-FTPD Format String Bug RPC.STATD Format String Bug	Linux	System Call Randomization
Lion	CAN-2001-0010	BIND 8 Buffer Overrun	Linux	Sys. Call Rand.
Slapper	CAN-2002-0656	OpenSSL 0.9.6d Buffer Overrun	Linux	Sys. Call Rand.
Malicious Web Site	MS05-002 CAN-2004-1305	Vulnerability in the Cursor and Icon Format Handling in IE	Windows	Enhanced ASLR (EAT Randomization)
Malicious Web Site	MS05-014 CAN-2005-0055	Heap Memory Corruption in IE DHTML method	Windows	Enhanced ASLR (Decoys + Guard Pages)
Malicious Web Site	MS05-020 CAN-2005-0053	Race Condition in IE DHTML Object Memory Management	Windows	Enhanced ASLR (EAT Randomization)
Malicious Web Site	MS05-025 CAN-2005-1211	PNG Image Rendering Memory Corruption in IE	Windows	Enhanced ASLR (Decoys + Guard Pages)
Zero-Day Exploit	MS05-037 CAN-2005-2087	IE JView Profiler Vulnerability (July 6, 2005)	Windows	Enhanced ASLR (EAT Randomization)
Zero-Day Exploit	MS05-052 CAN-2005-2127	Visual Studio .NET "msdds.dll" Remote Code Execution Exploit (August 17, 2005)	Windows	Enhanced ASLR (EAT Randomization)

Table 3. A Representative Subset of Code Injection Attacks Thwarted by RandSys

sys_getpeername system call on each file descriptor until the call succeeds and indicates that the peer TCP port is from the attacking machine. The system call randomization of RandSys effectively breaks the consistent static system call mapping in Linux and thus successfully corrupts the worm infection. More specifically, each worm infection is corrupted when the first system call, namely *sys_getpeername*, is attempted.

Effectiveness of enhanced ASLR randomization
The first two worm examples show the effectiveness of system call randomization. We next demonstrate the effectiveness of our enhanced ASLR techniques in defeating the MSBlast worm and the Sasser worm. Note that neither worm assumes static system call mapping. Instead, they leverage library APIs for their actions. More specifically, they first leverage the PEB data structure [10] to locate the *kernel32.dll* base address and then look up its EAT table to find the requested function name. As analyzed in

[10], the Sasser worm attempts to dynamically locate the following functions: *GetProcAddress*, *CreateProcessA*, *ExitThread*, and *LoadLibraryA*, from the *kernel32.dll* library. The *LoadLibraryA* function will be later invoked to load the *ws2_32.dll* library, which exports a number of basic networking-related library APIs, such as *bind*, *listen*, and *accept*. Our enhanced ASLR schemes, particularly library API randomization, randomize the EAT table entries, breaking the dynamic lookup process in the shellcode and thus successfully corrupting the infection. More specifically, each worm infection is corrupted when a function name resolution is attempted, which occurs at the beginning of the shellcode execution.

5.3 Thwarting Real-World Zero-Day Exploits

We have used RandSys against two zero-day exploits, each of which exploits an unpatched IE web browser vulnerability. As these two exploits are quite simi-

lar in both the nature of the vulnerabilities (JView Profiler vulnerability/MS05-037 and Microsoft Visual Studio .NET "msdds.dll" vulnerability MS05-052) and the exploitation means (*code-spraying* attacks), we only detail one exploit in the rest of this section. Figure 2 shows the malicious content of an "in-the-wild" exploiting web page, which takes advantage of the JView Profiler vulnerability (MS05-037) and utilizes the *code-spraying* attack as described below:
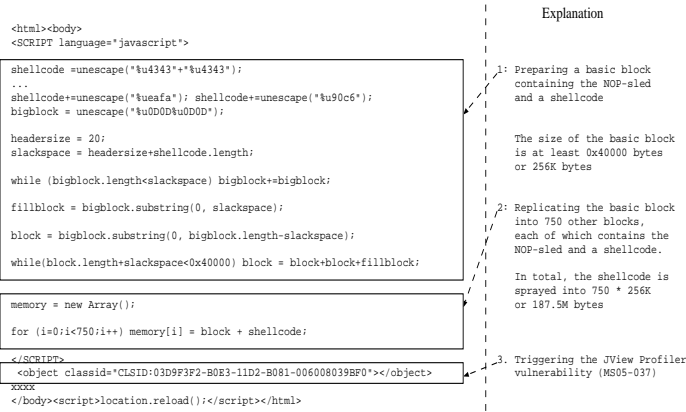
```
<html><body>
<SCRIPT language="javascript">

shellcode =unescape("%u4343"+"%u4343");
...
shellcode+=unescape("%ueafa"); shellcode+=unescape("%u90c6");
bigblock = unescape("%u0D0D%u0D0D");

headersize = 20;
slackspace = headersize+shellcode.length;

while (bigblock.length<slackspace) bigblock+=bigblock;

fillblock = bigblock.substring(0, slackspace);

block = bigblock.substring(0, bigblock.length-slackspace);

while(block.length+slackspace<0x40000) block = block+block+fillblock;

memory = new Array();

for (i=0;i<750;i++) memory[i] = block + shellcode;

</SCRIPT>
  <object classid="CLSID:03D9F3F2-B0E3-11D2-B081-006008039BF0"></object>
xxxx
</body><script>location.reload();</script></html>
```

Explanation

1: Preparing a basic block containing the NOP-sled and a shellcode

The size of the basic block is at least 0x40000 bytes or 256K bytes

2: Replicating the basic block into 750 other blocks, each of which contains the NOP-sled and a shellcode.

In total, the shellcode is sprayed into 750 * 256K or 187.5M bytes

3. Triggering the JView Profiler vulnerability (MS05-037)

Figure 2. An "In-the-wild" Malicious Web Page with the *Code-Spraying* **Attack**

(1) A javascript-based code snip in the malicious web page first prepares a basic memory block of 256K bytes containing a large NOP-sled (performing nop operations) and a particular shellcode. This block is then replicated to 750 other memory blocks. As a result, the shellcode (including the NOP-sled) is sprayed all over the allocated heap space of 187.5M bytes.

(2) The JView Profiler Vulnerability (MS05-037) is triggered, which results in the execution of the shellcode located somewhere in the allocated heap space. Note that existing ASLR schemes can make the actual location of the injected shellcode (contained in the allocated heap space) hard to predict. However, the code-spraying attack is able to *overcome* this challenge by populating the shellcode in a large memory space. As long as the overwritten code pointer (e.g., return address) points to somewhere inside this large memory space, the shellcode will eventually get executed.

(3) Once the shellcode is executed, it starts to unfold itself by performing an XOR operation. It then jumps into the middle of the unfolded shellcode body by skipping the first 16 bytes, which turns out to be the hash values of four different function names, i.e., "LoadLibraryA", "SetErrorMode", "ExitProcess", and "URLDownloadToFileA". These functions or their actual memory addresses need to be resolved before the exploitation can proceed. The first three functions are exported by the *kernel32.dll* module while the last one is exported by the *urlmon.dll* module.

(4) Next, the attack code attempts to locates the *kernel32.dll* base address by iterating the SEH [8] chain until the last SEH handler is located. Based on the facts that (i) the last SEH handler resides inside the *kernel32.dll* module and (ii) a module is always aligned on 64K-byte boundaries, the code uses the last SEH hander as a starting point for walking down with an increment of 4K bytes. A check is performed to see if the two characters at that point are "MZ", which usually marks the MSDOS header. Once a match is found, it is assumed that the base address of *kernel32.dll* has been located.

(5) Finally, this base address is used to parse the PE file format to locate the EAT name table. Each name entry within the EAT table is checked to locate those intended function APIs, such as "LoadLibraryA", "SetErrorMode", and "ExitProcess".

Under RandSys, the EAT names have been randomized. As a result, the exploitation is effectively thwarted at step (5) described above. The new "spray-and-hit" strategy of code-spraying attack also demonstrates the unique advantage of RandSys over the ASLR scheme.

6   Related Work

Building diversity into networked computers for better security was first advocated by Forrest et al [22]. Recent work has applied the same diversity principle to code-based instruction set randomization (ISR) [12, 13, 24, 33] and memory-based layout randomization (ASLR) [2, 14, 15, 37]. ISR makes the "working" instruction set hard to predict, and is able to foil the execution of injected machine instructions. However, it is vulnerable to attacks that avoid using injected machine instructions, such as *return-into-libc* and *return-into-dl* attacks. ASLR randomizes the memory layout and is robust against attacks that hijack predetermined specific memory addresses. However, it is susceptible to *code spraying* and *address spraying* attacks which avoid using specific memory locations. Recalling the code spraying example described in Section 5.3, the attack code prepares a large heap space ($750*256K$ bytes) and then fills it all over with the intended shellcode. After that, the attacker only needs to guess the location of a shellcode replica with a probability of $750*256K/2^{32} = 4.6\%$, which contains a very low entropy (4 bits if taking into account that the Windows kernel occupies the upper half memory space). Note that ASLR is fundamentally susceptible to such spraying attack: not only in current 32-bit architecture, but also in the next-generation 64-bit architecture. By effectively and practically combining both ISR and ASLR, RandSys is able to defeat these attacks fundamental to each of ISR and ASLR individually.

Non-Execute (NX) [1, 23, 35] protection support from both hardware vendors (such as Intel and AMD) and operating system providers (e.g., W^X support in OpenBSD and Data Execution Protection from Microsoft) pro-

	Example Attack Categories			
	Regular code injection attacks (e.g., stack-smashing attacks)	*return-into-libc* attacks	*return-into-dl* attacks	*code-spraying* attacks
ISR	√	×	×	√
ASLR/PaX	√	√	×	×
Non-eXecute	√	×	×	√
RandSys	√	√	√	√

Table 4. Comparison of RandSys with Other Protection Approaches

vides page-level memory protection (read, write, or execute) and renders the injected machine instructions *non-executable*. Similar to ISR, NX fails to cope with attacks which avoid using injected machine instructions, including *return-into-libc* and *return-into-dl* attacks.

Table 4 summarizes the unique position of RandSys in relation to ISR, ASLR (with PaX [2] as an representative ASLR example), and NX.

Chew et al [17] described an operating system-based randomization approach, which not only provides basic memory space layout randomization, but also attempts to system-wide re-number system calls. Note that the notion of system call re-numbering [17] is close to the system call randomization in RandSys. However, there are a number of fundamental differences: Their re-numbering is implemented by recompiling the kernel with a different but another *fixed* system call mapping. As a result, any re-mapping attempt requires the physical machine rebooting, and the re-mapping is achieved at the granularity of machines — different processes still have the same system call mapping. In contrast, RandSys establishes a unique system call mapping for each individual process at its creation time. In addition to system call number randomization, RandSys also provides an enhanced ASLR protection.

In addition to the randomization efforts to counter code injection attacks, various other techniques [11, 18, 19, 20, 21, 25, 28, 31, 36] are also proposed to address this attack. Broadly speaking, static analysis techniques [21, 28, 36] attempt to statically analyze program source code to discover possible vulnerabilities, while dynamic analysis techniques [11, 18, 19, 20, 25, 27, 31] leverage run-time information to dynamically detect or confine possible attacks. By comparison, like ISR and ASLR, RandSys introduces diversity into existing computer systems in the first place, which is attack- or vulnerability-independent.

7 Conclusion

In this paper, we have presented RandSys, a novel system that effectively combines Instruction Set Randomization (ISR) and Address Space Layout Randomization (ASLR). This combination allows RandSys to defeat attacks fundamental to each of ISR and ASLR individually. Another contribution of our work is that we randomize

only system-call instructions rather than the entire instruction set, hence effectively address the performance problem of ISR. We have also developed new techniques that make control flow hijacking extremely difficult, including decoys, guard pages, independent randomization for both import and export tables, as well as a defensive execution scheme that detects shellcode-contained pages. We have implemented and evaluated RandSys for both Linux and Windows. Our experiments show that RandSys can effectively thwart a wide variety of code injection attacks on the Internet with a small overhead.

References

[1] Intel: Execute Disable Bit Functionality That Combats "Buffer Overflow" Attacks. `http://www.intel.com/business/bss/infrastructure/security/xdbit.htm`.

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

[3] Code Red Worms. *CAIDA Analysis of Code-Red Worms* `http://www.caida.org/analysis/security/code-red/`, 2001.

[4] Linux Lion Worms. `http://www.whitehats.com/library/worms/lion/`, 2001.

[5] MSBlaster Worms. *CERT Advisory CA-2003-20 W32/Blaster Worms* `http://www.cert.org/advisories/CA-2003-20.html`, August 2003.

[6] Sasser Worms. `http://www.microsoft.com/security/incident/sasser.asp`, May 2004.

[7] Websense Security Labs: Security Trends Report (Second Half 2004). `http://www.websensesecuritylabs.com/resource/WebsenseSecurityLabs20042H_Report.pdf`, 2005.

[8] Windows Platform SDK: Structured Exception Handling. `http://msdn.microsoft.com/library/en-us/debug/base/structured_exception_handling_functions.asp`, July 2005.

[9] Zotob Worm Hits CNN and Goes Global. `http://it.slashdot.org/article.pl?sid=05/08/16/2247228\&tid=220\&tid=188`, August 2005.

[10] Xuxian Jiang, Helen J. Wang, Dongyan Xu, and Yi-Min Wang. RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. *CERIAS Technical Report 2007-31, Purdue University*, 2007.

[11] 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.

[12] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *In Proceedings of the 10th ACM CCS, Washington, DC*, October 2003.

[13] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *ACM Transactions on Information and System Security*, 2005.

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

[15] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *Proceedings of the 14th USENIX Security Symposium 2005 , Baltimore*, August 2005.

[16] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD*, August 2005.

[17] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. *CMU Technical Report CMU-CS-02-197*, December 2002.

[18] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic Protection from Printf Format String Vulnerabilities. *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[19] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities. *Proceedings of the 12th USENIX Security Symposium, Washington, DC, USA*, August 2003.

[20] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, , and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *Proceedings of the 7th USENIX Security Symposium, San Antonio, Texas, USA*, January 1998.

[21] D. Evans. Static Detection of Dynamic Memory Errors. *In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[22] S. Forrest, A. Somayaji, and D. H. Ackley. Building Diverse Computer Systems. *Workshop on Hot Topics in Operating Systems, pages 67-72*, 1997.

[23] Gaurav S. Kc and Angelos D. Keromytis. e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing. *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC), Tucson, AZ*, December 2005.

[24] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. *In Proceedings of the 10th ACM CCS, Washington, DC*, October 2003.

[25] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. *Proceedings of the 11th USENIX Security Symposium, San Francisco, USA*, August 2002.

[26] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating Mimicry Attacks Using Static Binary Analysis. *Proceedings of the 14th USENIX Security Symposium 2005 , Baltimore*, August 2005.

[27] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting Against Unexpected System Calls. *Proceedings of the 14th USENIX Security Symposium 2005 , Baltimore*, August 2005.

[28] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *In ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2004.

[29] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Volume 0x0b, Issue 0x3a*, December 2001.

[30] Frederic Perriot and Peter Szor. An Analysis of the Slapper Worm Exploit. *Symantec White Paper* `http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf`.

[31] Niels Provos. Improving Host Security with System Call Policies. *Proceedings of the 12th USENIX Security Symposium, Washington, DC, USA*, August 2003.

[32] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address Space Randomization. *In Proceedings of the 11th ACM CCS, Washington, DC*, October 2004.

[33] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. *In Proceedings of the USENIX Annual Technical Conference, pp. 149 - 161, Anaheim, CA*, April 2005.

[34] Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? The Effectiveness of Instruction Set Randomization. *Proceedings of the 14th USENIX Security Symposium 2005 , Baltimore*, August 2005.

[35] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux. `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, August 2004.

[36] 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*, February 2000.

[37] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. *In Proc. of 22nd Symposium on Reliable and Distributed Systems, Florence, Italy*, October 2003.