Zeus Software Defender Technology

White Paper Version 1.0

July 31, 2020

Zeus SW Defender, LLC

Contents

SUMMARY

Compilation of C and C++ Programs with Static and Dynamic Encryption of Code Pointers

Protecting computer memory is critical in program security because most of the breaches in programs written in C and C++ start in memory. Among the program components in memory, code pointers have drawn exceptional attention because they store control data. When code pointers are accessible from outside for a malicious intent, they pose vulnerabilities such as control-flow interception and control data leaks. Zeus Software Defender Technology (Zeus for short) integrates *code pointer encryption* into compilers and toolchains and produces executables hardened against the vulnerabilities. Zeus further reinforces the encryption of return addresses in the stack frame by *dynamic reencryption*, which renews encryption states during program execution. Zeus' example use cases are to prevent cyber security breaches through the buffer overflow attack of computer memory, return-oriented programming, and control information leak attacks. Benchmarking of C programs with O3 optimization of the recent implementation shows overheads of 7.66 % in execution time for CPU-intensive and 3.52 % for IO-intensive workloads respectively. Zeus is currently in progress to improve performance and coverage.

I. INTRODUCTION

Protecting computer memory is critical in program security because most of the breaches in programs written in C and C++ start in the memory.[1] A typical instance is the buffer overflow attack. Adversaries supply long input that overrides a stored return address next to a buffer in the runtime stack. As a result of the overwriting, the subsequent return instruction diverts control-flow where the malicious code starts.[2]

For example, an indirect branch instruction loads the program counter with its operand control data. Then program execution continues to a specific address that the control data represent. The memory locations storing control data are collectively called *code pointers*, and the return addresses[3] are the most common *code pointers*, which are addresses of instructions. Code pointers pose critical vulnerability when it is possible to access them by malicious users.

Researchers have pursued the mitigations of code pointer vulnerabilities for more than three decades. Among them, control-flow monitoring discerns compromised control data by detecting unexpected control transfer. Another noticeable approach obfuscates program components by randomizing their placements or contents, that is, the bits constituting their values. The randomization of the components' bits usually encrypts critical data and decrypts them when an instruction consumes the data. Since encryption-based schemes focus on crucial data and their locations that take only a small portion of programs' data area, they have lower costs than monitoring and other randomization if the granularity levels are comparable to each other. Mitigation methods may set granularity levels at instructions, basic blocks, functions, or segments.

However, encryption-based solutions still need improvements in protection coverage and resistance against information leaks. Zeus addresses these two issues with an endeavor to reduce overhead execution times. C and C++ programmers tend to consider performance ahead of security because use cases of C and C++ programs belong to the area where high-performance matters. Zeus integrates into program compilation, linking, loading, and the early stage of execution. Zeus

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," 2013, pp. 48–62. IEEE, 2013; R. Seacord, "Secure Coding in C and C++, Second Edition," Chapters 2, 3, 4, Addison-Wesley, 2013.

[2] For example, the return address has changed to the starting address of a shellcode or return-oriented programming (ROP) chain.

[3] The "return address" indicates either the location or the value at the location in this document.

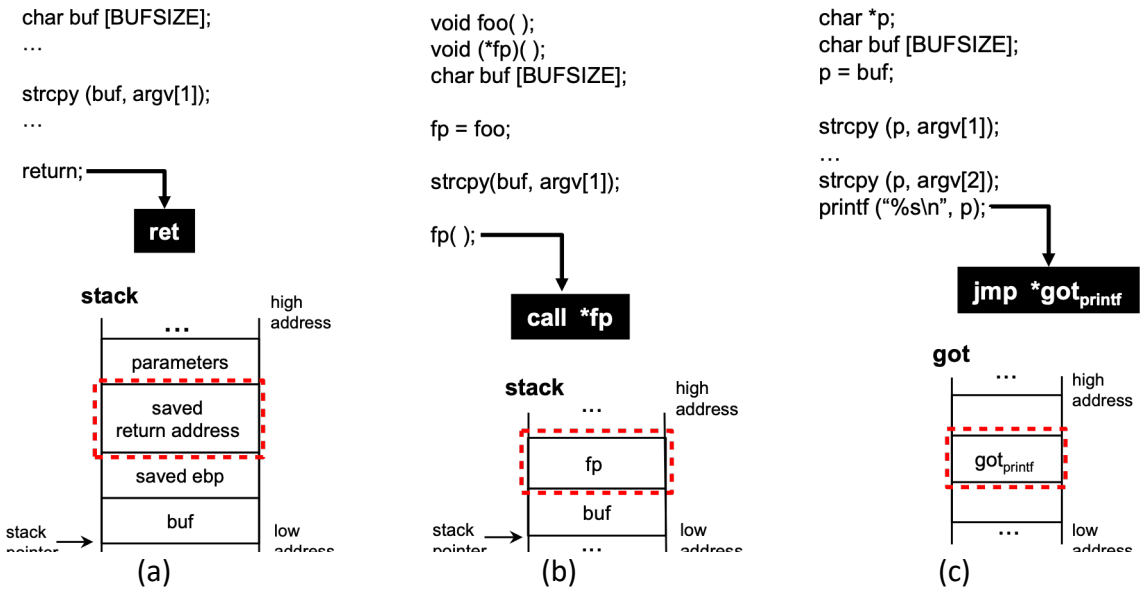

Figure 1 Code pointers and their vulnerabilities

produces executables that randomize code pointers with reinforcement against their leaks. The following sections present a technical overview of Zeus and test results of effectiveness and performance benchmarking.

II. VULNERABILITIES OF CODE POINTERS

A code pointer is vulnerable to overwriting when they are accessible from outside of a program. When the code pointer is next to an array used as a buffer to which the program copies input, outside adversaries can overwrite the code pointer beyond the array bound, modifying the code pointer. Then the execution of an indirect jump referencing the pointer continues to the address that the modified control data designates. This section distinguishes code pointers according to their usage context, vulnerabilities, and mitigations of their defects.

2.1 CLASSIFICATION OF CODE POINTERS

Code pointers can be classified into four groups according to their usage context. The first and most common code pointers dwell in the runtime stack and accommodate return addresses. The second group is the function pointer variables appearing as literals and declared in source programs. They are called *function designators* in the C language definition. The third group's code pointers belong to compiler-generated tables referenced for linking shared libraries. Return

instructions, indirect function calls, and invocations of shared library functions reference these three groups of code pointers correspondingly. The fourth group is function pointer tables attached for initialization or finalization by compilers and toolchains. Their shapes are similar to the third group's, and their usage, to the second group's.

2.2 Vulnerability Instances of Code Pointers

Let's consider the potential vulnerabilities of each code pointer type as per the classification. Zeus treats the last group in the same way that it does the second, and thus we cover the first three kinds and do not discuss the last one separately. Figure 1 shows how to compromise each group of code pointers through buffer overflow attacks. Other methods for compromising code pointers such as format strings and arbitrary writes can accomplish the same goal.

Return instructions are of the most prevalent indirect branch instruction type at runtime, taking 85% of indirect branches in SPEC CPU2006 benchmark. Aside from frequent occurrences, their predictable locations in runtime stack frames make them easily exploitable targets. Figure 1(a) presents an example relevant to the vulnerability of the saved return address. The invocation of a function pushes a return address onto a stack frame. The saved return address points to the location of the instruction following the call. A malicious user can overflow the array `buf` to overwrite the saved return address by providing the function `strcpy` with a string argument long enough to reach the return address.

The code in Figure 1(b) shows the vulnerability of a function address in a function pointer `fp`. Function designator literal `foo` is not vulnerable, but it becomes weak when it moves into function pointer variable `fp`. Because function `strcpy` does not check the size of the destination array `buf`, an attacker can overwrite the function pointer `fp` with an oversized input in `argv[1]` so that the subsequent call `fp()` transfers control to an address specified by the attacker. Vulnerable function pointers can be in any writable segment such as the stack, heap, or BSS.

Figure 1(c) shows an unsafe code that references shared libraries' functions, such as `printf`. A call to function `printf` transfers control to its procedure linkage table (PLT) entry, which calls the dynamic linker to relocate the global offset table (GOT) entry to the address of `printf`.[4] After adjustment, the indirect branch instruction of the PLT transfers control to the real site of `printf`

[4] The GOT and PLT are terms of the ELF binary format. Still, the code pointer classification is valid to other binary formats such as COFF and PE to which Zeus is also applicable.

in the shared library by dereferencing the adjusted GOT entry. Attackers can intercept the target program's control by overwriting the GOT entry in the same way as overwriting the function pointers in Figure 1(b). In Figure 1(c), with tainted `argv[1]`, the first call to `strcpy` compromises pointer variable `p` by overflowing `buf`. To exploit the vulnerability, a malicious user can make pointer `p` refer to $\mathbf{got}_{\texttt{printf}}$. With `argv[2]` compromised to hold malicious code address, the subsequent call to `strcpy` overwrites $\mathbf{got}_{\texttt{printf}}$ with the address of the malicious code. Finally, the invocation of `printf` starts the malicious code.

2.3 MITIGATIONS OF CODE POINTER VULNERABILITY

Control-flow monitoring, one of the two streams protecting code pointers, aims to detect unexpected control-flow. The monitoring identifies irregular control-flow by comparing runtime control transfer with control-flow graphs (CFGs) constructed by static analyses. Researchers have developed numerous monitoring-based solutions, but few have survived in the market because they have barely reached the acceptable effectiveness with bearable overhead execution times. These drawbacks in accuracy and performance result from static CFGs that cannot but include many unrealistic branches. Developers have traded off security for efficiency by increasing the sizes of monitoring granularities, and adversaries have not missed the expanding loopholes in the weakened protective cover.

The second stream of code pointer protection obfuscates programs by randomizing program components' placement or critical data. Randomization makes it hard to modify code pointers as intended without knowing the precise way of de-randomization. Encryption using secret keys, powerful and efficient randomization, requires corresponding decryption with the same or matching keys before dereferencing the code pointers. Otherwise, programs crash thwarting further progress. Though this approach has low execution overhead, traditional randomization method has shown two shortcomings, which we will go over in the following sections.

2.4 SHORTCOMMINGS WITH RANDOMIZATION: NO COVERAGE OF FUNCTION POINTERS

There have been attempts to perform a crypto operation for function pointer protection. This approach encrypts a function pointer after assigning a control data to the pointer and decrypts it before reading from the pointer. With pointer tracking, the program of Figure 2 encrypts function pointer variable `fp` after assigning `foo` to `fp`. The next statement decrypts `fp` before reading it and encrypts another pointer `fp1` after copying from `fp`. Finally, the program decrypts `fp1`

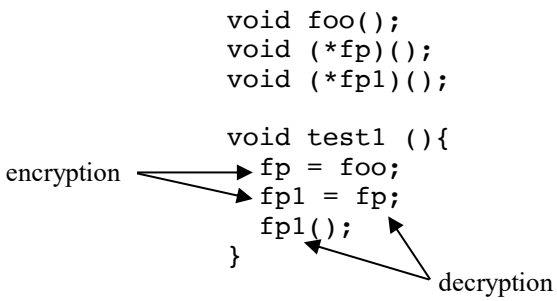

Figure 2 Program with function pointers

before the indirect call referencing `fp1`.

As long as function pointer variables are the subject of encryption, encryption-based approaches would hardly get to any practically working version because it requires pointer tracking, which does not produce an exact solution in general. C and C++'s liberal type casting further aggravates the difficulty. It is hard to identify precisely which pointer variables and their possible aliases to encrypt. Limiting encryption to the group of *must-alias* pointer variables results in sparse coverage while extending to *may-alias* pointers could result in a severe performance drop. More seriously, neither of them can guarantee correctness. *PointGuard*[5] attempted the manipulation per pointer variable, but there has been no successful implementation report yet. This approach with perfect compatibility and acceptable overhead remains challenging, probably an impossible task for C and C++ programs.

2.5 SHORTCOMMINGS WITH RANDOMIZATION: CONTROL INFORMATION LEAKS

Every encryption-based randomization, in general, has the vulnerability of information leaks. Risks of information leaks are concerned with not only encryption keys but also code pointers. Once keys leak, adversaries can craft exploits for further attacks. Although a program does not employ code pointer encryption and has no key to disclose, the leaks from code pointer are still worthwhile for designing exploits. Sometimes, adversaries do not even need to break encryption keys. They intercept control-flow by overwriting a code pointer with the disclosed from another code pointer encrypted with the same key. Cryptographically robust algorithms and preservation of encryption keys' confidentiality are necessary to mitigate the leaks. Nevertheless, code pointer encryption cannot adopt secure encryption methods if their execution times cost highly.

[5] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. "PointGuard ™: Protecting Pointers from Buffer Overflow Vulnerabilities." In *Proceedings of the 12th Conference on USENIX Security Symposium*, 12:91–104, 2003.

2.5.1 Weak Encryption Primitive

Many encryption schemes adopt the XOR instruction as the crypto primitive for encryption and decryption because of its ubiquity in microprocessors and low overhead execution time. The XOR instruction efficiently randomizes each bit independently of others. However, XOR encryption is so brittle that if an adversary has disclosed a return address *r* in plaintext and the corresponding ciphertext *e* from the target, the encryption key *k* is XOR(*r*, *e*) as the inverse of the XOR is itself. Despite its weakness, there is no better choice other than the XOR in the current instruction set architectures because programmers are reluctant to adopt a high-overhead solution, no matter how secure it is. Preference for performance over security is difficult to reverse in the C and C++ programming community at present. As a result, solution providers choose an instruction from the existing instruction set for crypto operation, and the XOR is frequently the most suitable one.

2.5.2 Lack of Diversity In Encryption Keys

If keys are not diverse, attackers can disclose a key and exploit it at other times or locations. The attack sequence of leakage followed by a control-flow interception at different times and locations becomes feasible when target programs are resilient. Typical resilient applications are web servers that fork a child process whenever a new request arrives. Under the condition of resilience and lacking keys' diversity, adversaries can disclose the encrypted control data by *brute-force incremental byte reading.* It overwrites a byte of code pointer, such as a return address in a stack frame, with a random byte and checks if the target program crashes. A side-channel analysis, such as the timing analysis, discerns whether the target is alive. If the victim dies, the random byte does not match with the target byte. Otherwise, the guessed byte is probably correct. Though the target process crashes because of overwriting with an incorrectly guessed byte, its parent would spawn a new child, which allows repeated byte reading.

III. Zeus' Innovative Methods For Code Pointer Protection

3.1 Static Encryption of Code Pointers

Zeus integrates into a compiler, static linker, dynamic linker, and the standard C library. Zeus-integrated compiler and toolchain cooperate to insert code snippets into programs' security-critical locations while building executables. At runtime, the injected code fragments carry out *static*

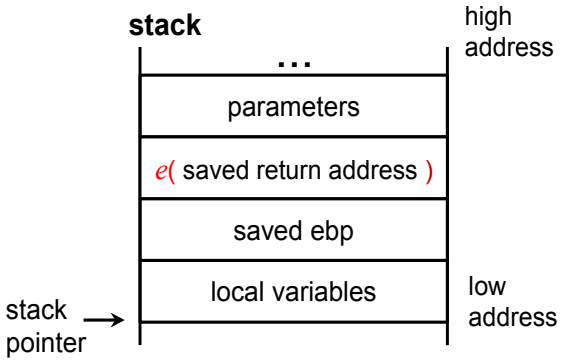

Figure 3 Layout of a stack frame. *e*(saved return address) denotes an encrypted return address after the execution of a call instruction

encryption for all sorts of code pointers and *dynamic reencryption* additionally for return addresses of the runtime stack. Static encryption does not change its states at runtime, whereas dynamic reencryption does. The hardened binaries mitigate the flaws of the encryption-based randomization in the preceding section. This section explains *static* code pointer encryption, emphasizing function pointer protection, and the next one, dynamic reencryption of return addresses.

3.1.1 Encryption of Return Addresses in the Runtime Stack

In the x86 computer architecture, invoking a function pushes a return address onto the runtime stack. The return address stays in the stack until a return instruction pops out the address from the stack. Zeus attaches an encryption sequence at the beginning of the function prologue and a decryption sequence at the end of the function epilogue before a return instruction. Each return address has a privatized encryption key to prevent attacks from exploiting a leaked key somewhere else. As a result, Zeus-built executables encrypt a return address right after its birth, as shown in Figure 3, and restore its plaintext before going back to the call site and terminating its lifetime. The encrypted return address remains fixed throughout its lifetime. The matching encryption and decryption protect the return address' integrity and confidentiality.

3.1.2 Encryption of Function Designators

Zeus-built binaries avoid working directly on function pointers to encrypt or decrypt. Instead, they encrypt the addresses that the function pointers accommodate. In Figure 4, the program encrypts the address that function designator `foo` represents instead of the function pointer variable `fp`. Then the encrypted `foo` is assigned to variable `fp`, and the assignment of `fp` to `fp1`

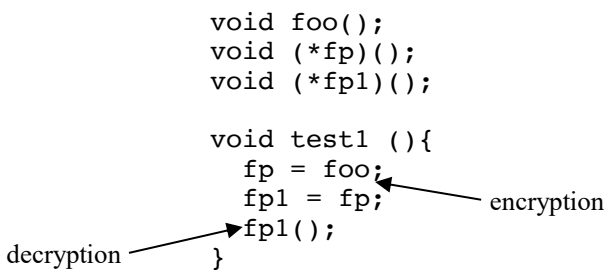

Figure 4 Function designator encryption

also occurs without any crypto work. Function pointers always hold encrypted function addresses. Therefore, an encrypted function address may move from one pointer to another without decryption and encryption for reading and writing the address from or to code pointers.

The shift from code pointers to their control data makes it simple to protect function pointers by encryption. If function pointers do not appear in arithmetic expressions, control data encryption eliminates encryption and decryption applied to function pointer variables and obviates pointer alias analysis. Also, encryption and decryption timings become clear. Finally, randomization costs reduce when compared with the costs managing function pointers per function pointer reference. For instance, let us assume that function `test1` in Figure 4 repeats ten times. Function designator encryption necessitates eleven crypto operations of one encryption and ten decryptions, whereas the pointer variable encryption requires forty of twenty encryptions and twenty decryptions. See Figure 2.

Note that function designator encryption may have a compatibility issue. The call using variable `fp2` in Figure 5 would cause a crash under function designator encryption. The program casts the function designator's type to integer type, performs an arithmetic operation, and executes a call instruction after casting back to the function pointer type. The C standard specifies that such code fragments in Figure 5 have undefined behavior, and Zeus leaves defining its behavior to programmers. They may manually insert function calls of encryption and decryption to fix the

```
void foo();
void (*fp2)();

void test2() {
  fp2 = foo + 0xa;
  fp2();
}
```

Figure 5 Program with arithmetic operation on a function pointer

behavior. We have an interim affirmative result that Zeus has succeeded in building the standard C library without manual insertion of crypto operations for undefined behavior. Thus we anticipate there is practically no obstacle to incorporating in compilers function designator encryption.

3.1.3 FUNCTION POINTER TABLE ENCRYPTION FOR DYNAMICALLY LINKED LIBRARY

Programs invoke dynamically linked shared library functions through the PLT. Each PLT entry calls the library's function using an indirect jump instruction referencing the corresponding GOT entry. A GOT entry for the PLT is a code pointer holding a function's runtime address. The dynamic linker of the standard library is responsible for the encryption of the GOT function pointers.

3.1.4 ENCRYPTION OF MISCELLANEOUS FUNCTION POINTER TABLES

Compilers and toolchains generate function pointer tables to facilitate initialization or final cleaning job. Zeus processes the table entries in the same way as the function designators in Section 3.1.2.

3.2 DYNAMIC REENCRYPTION OF RETURN ADDRESSES

Static encryption protects code pointers, but its effectiveness against information leaks drops significantly for resilient programs, for which unlimited brute-force attacks are feasible. *Crash-resistance oriented programming* (*CROP*)[6] is a brute-force attack exploiting programs' resilience. Zeus mitigates the weakness of CROP by dynamically diversifying keys. In addition to providing a privatized encryption key with each return address, Zeus-built binaries reencrypt return addresses during execution with new keys whenever control-flow enters risky regions. The diversity of encryption keys changing along with locations and times makes the leaked bits garbage.

Ideally, return addresses would be reencrypted before every access to memory, but frequent reencryption would incur significant overhead. Besides, extremely frequent reencryption is unnecessary because not all memory accesses are vulnerable. At present, Zeus activates runtime reencryption of return addresses before or after risky operations, including calling to copy functions, spawning a process or thread, and handling exceptions. The following elaborates Zeus' design choices of reencryption timings for return addresses.

---

[6] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz, "Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding," In Proceedings of *The Network and Distributed System Security Symposium*, San Diego, CA, USA, Feb. 2016,

3.2.1 REENCRPTION BEFORE RISKY FUNCTION CALLS

Many C Standard Library functions, such as *strcpy*, *memcpy*, and *strdup* copy bytes from one memory location to another without checking the destination's size. When these functions copy data, attackers can force them to overwrite return addresses in the stack. Runtime reencryption precludes attackers from exploiting a disclosed key when they attempt overwriting return addresses because the key is valid only for a specific return address in a limited interval. Zeus currently inserts code fragments for reencryption before calls to 60 copy functions classified as risky. Zeus can easily enroll new insecure functions subject to reencryption.

3.2.2 REENCRYPTION AFTER SPAWNING PROCESSES AND THREADS

Because a process inherits memory layout from its parent under the conventional process forking, both the parent and its child processes share the same encryption keys and encrypted return addresses. If one of the processes leaks a key, an adversary can attack the other processes by exploiting the key. Zeus generates new keys to prevent key sharing among processes or threads from a common ancestor and reencrypts the spawned child processes' or threads' return addresses. Thus, all return addresses have different encryption keys across processes. It is unlikely to disclose keys, but the diversity makes it hard to exploit disclosed keys and control data even when they leaks.

3.2.3 REENCRYPTION CONFORMING TO STACK UNWINDING

Throwing an exception initiates stack unwinding that deallocates stack frames and destroys any data object linked to the frames. Activating *longjmp* or *siglongjmp* also causes stack unwinding. Unwinding continues until the control-flow reaches an exception handler or a location marked by *setjmp* or *sigsetjmp*. As stack unwinding requires plaintext return addresses in the stack, Zeus maintains compatibility with stack unwinding by decrypting all randomized return addresses in the stack before unwinding. After unwinding, Zeus reencrypts the remaining return addresses back with new keys.

3.2.4 VARIATIONS OF REENCRYPTION FOR BETTER PERFORMANCE

Zeus has variations of dynamic reencryption for better performance. When the runtime stack rapidly expands as might happen in recursion, reencrypting all return addresses may increase execution time beyond tolerance. Zeus allows programmers to tradeoff between performance and security, without sacrificing either. The first variation reencrypts only the most recent return

Table 1 Protection efficacy over other schemes per RIPE benchmark test

Protection	Prevention (%)
No protection	0
Libsafe	7
LibsafePlus	19
StackShield	36
ProPolice	40
LibsafePlus+TIED	77
CRED	79
Non-executable Stack	89
*Zeus	100

address in the top stack frame. In general, Zeus may apply reencryption to $k \geq 1$ most recently pushed or randomly selected return addresses. Limiting the number of reencrypted return addresses limits overhead to a fixed constant time.

Another variation works with a single key for all cases of reencryption per thread and process. The mono key option reduces the managerial work for encryption keys. The encryption of all return addresses relies on a single key, and thus, it is compulsory to update the encryption states of all return addresses whenever the key changes. The key is always in a dedicated register to eliminate the risk of leakage due to register spills.

3.2.5 Extension for Other Code Pointers

Dynamic reencryption applies to the function pointers of the GOT. We are working on it.

IV. Zeus' Effectiveness and Performance of Code Pointer Protection

4.1 Mitigations of Buffer Overflow Attacks and Control Interception

To evaluate Zeus' efficacy, we tested Zeus with three real-world attacks that CVE[7] lists. Zeus successfully mitigates attacks in each intrusion scenario. Encryption and reencryption of code pointers provide proper protections.

- **Proftpd-1.2.7** CVE-2003-0831. Proftpd is a popular FTP server. It has a vulnerability in the part downloading files. A buffer overflow may exploit the vulnerability to overwrite a pointer in heap memory.

[7] Common Vulnerabilities and Exposures. https://cve.mitre.org

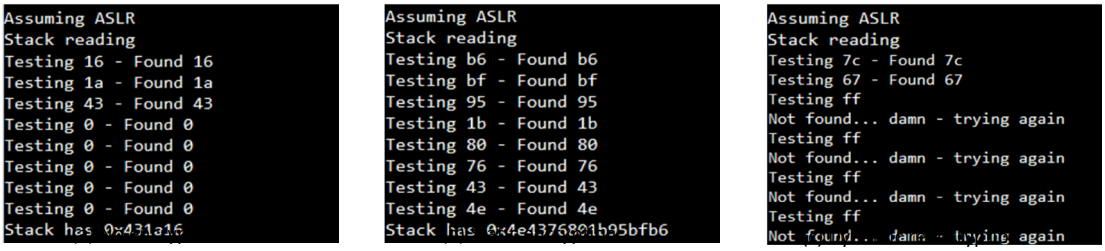

Figure 6 Protection of Ngnix against BROP using Zeus. Incremental brute-force memory reading discloses a return address when encryption is neither in effect nor dynamic.

- **Snort-2.4.2** CVE-2005-3252. This program is an intrusion prevention and detection system. It has vulnerabilities in the stack, and a buffer overflow attack can overwrite a return address and transfer control to injected code.
- **Samba-2.2.1** CVE-2003-0201. Samba is the standard Windows interoperability suite for Linux and UNIX programs. A malicious user can overflow a stack buffer with an unbounded string to overwrite a return address in the stack.

Given this knowledge, we expanded the RIPE benchmark test suite[8] and tested Zeus with diverse attack patterns using the suite. The expansion includes the attacks on the function addresses of the GOT. Table 1 summarizes the experiment results. Except for Zeus, the figures in the table come from the original RIPE paper. Unlike other mitigation schemes, Zeus thwarts the whole 850 cases and the lately added.

4.2 Protection Effectiveness against Information Leak Attacks

We tested whether Zeus could protect a resilient program from information leak attacks. We replicated CVE-2013-2028 that describes the Blind ROP (BROP) in the web server Nginx-1.4.0. The BROP starts by disclosing a return address and continues the next stage attack using the leaked address. We confirmed that programs without any protection or only with static encryption could not prevent return address leaks, as shown in Figure 6(a) and (b). Dynamic reencryption contrastingly prevents the exploit from incrementally reading more than two bytes of the encrypted return address, as demonstrated in Figure 6(c). The leaked bytes are reencrypted before brute-forcing the next byte. Attackers should guess all 6 bytes of a return address simultaneously, whose probability converges to zero, $1/2^{48}$.

[8] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen. "RIPE: Runtime Intrusion Prevention Evaluator." In Proceedings of the 27th Annual Computer Security Applications Conference, Orlando, FL, USA, 2011, pp. 41–50.

Table 2 Overheads in execution for CPU-bound applications. Benchmarks are SPEC2006 CPU. FDE stands for function designator encryption.

	Return address reencryption	Library rebuilding (*no*), %		Library rebuilding (*yes*), %	
		FDE (*no*)	FDE (*yes*)	FDE (*no*)	FDE (*yes*)
C	All frames	5.93	7.66	7.62	9.20
	Top frame	5.70	7.18	7.56	8.90
	Single key	5.80	7.20	7.55	8.92
C++	All frames	15.28			
	Top frame	15.07			
	Single key	15.11			

4.3 Overheads of Code Pointer Encryption

We measured overhead execution times with several different criteria. The first criteria is whether the benchmark are CPU-bound or IO-bound workloads. The second and third criteria are whether or not to include function designator encryption and rebuild C libraries with the Zeus-integrated compiler. The fourth criteria differentiates among the dynamic reencryption and its two variations. The final fifth applicable only to CPU-bound benchmarks distinguishes between C and C++ programs. The compiler option was O3 optimization for every benchmarking.

Table 2 summarizes percentile overhead execution times for SPEC2006 CPU. They increase by 5.93 % for reencryption of return addresses, and the overheads increase up to 7.66 % by function designator encryption. Rebuilding the standard C library increases the execution time by 1.54 % further. The two variations of dynamic reencryption did not show noticeable differences.

The execution times increase by 15.28 % for C++ programs. The stark increases come from calls to small size functions, constructors, and destructors. Since there is no available C++ library that the LLVM can compile, we could not incorporate function designator encryption into the toolchain for C++ programs. Two variations of dynamic reencryption show no significant difference, and the overhead reduction is marginal at present.

For performance benchmarking of IO-bound programs, we adopted HTTP servers Nginx and Apache. It is not appropriate to measure overhead execution times for perpetually running HTTP servers. Instead, we measured the round-trip times (RTTs) of processing a request with various configurations of a server and client in a single host to avoid network delay. We configured an HTTP server launches *W* workers, and a client generates *C* concurrent requests, where *W* and *C*

Table 3 Overhead round-trip time of web servers. Figures are geometric means of percentile increases of RTTs of Nginx and Apache for various configurations.

	Return address reencryption	Library rebuilding (*no*), %		Library rebuilding (*yes*), %	
		FDE (*no*)	FDE (*yes*)	FDE (*no*)	FDE (*yes*)
G-means, Nginx and Apache	All frames	2.93	3.52	3.61	3.90
	Top frame	2.56	3.25	3.24	3.29
	Single key	2.53	3.27	3.33	3.64

are powers of two less than or equal to 64, giving 49 measured RTTs. Table 3 shows the overheads in varying configurations. Figures are geometric means of percentile increases of RTTs of Nginx and Apache for various configurations.

The code pointer encryption seldom affects web servers' performance in network environments. The average RTT increases by 0.48 % when we set up two separate hosts for a server and client. Because network delays are unpredictable and larger than overhead execution times on the hosts, we anticipate that delays hide the execution times increased by Zeus.

V. ZEUS' MERITS AND LIMITATIONS

5.1 MERITS

Zeus can refine reencryption timing independently of other program components and without kernel support. A thread can reencrypt the return addresses in its stack while another continues execution in the same process. This feature can prevent attacks resorting to in-process crash-resistance by exploiting exception handling and multithreading. On the contrary, mitigations relying on kernel support cannot apply rerandomization freely because of its cost. As a typical instance, the dynamic descendants of *address space layout randomization* (ASLR) requires two times of context switching to and from a kernel. The kernel pauses a process, randomly relocates its components, and finally grants the process to resume execution. If rerandomization starts at an arbitrary point as in Zeus, the overhead increases further. Thus, rerandomization takes place only when the kernel cannot but take or release control for services. Process forking is a suitable time to rerandomize. We also expect that ASLR's family has difficulty in real-time applications because it is hard to estimate when and how long kernel's intervention would take.

Zeus has the advantage over the ASLR and its dynamic versions in maintaining relevant data.

ASLR family must relocate numerous components and adjust pointers accordingly. To minimize relocation workloads, the ASLRs generates position-independent code (PIC). However, PIC relies on indirect branches, so it has more pointers to protect than non-PIC. It is not desirable to increase the number of pointers because they can be adversaries' primary targets.

Zeus installs a data structure, called a *shadow stack*, for key management in the user space, hides it by random placement similar to the ASLR, and initializes surrounding locations by random bits. Besides, Zeus' dynamic reencryption has the effect of rerandomizing the shadow stack as well. Although its location may leak, the changing encryption states make it hard to disclose its contents.

5.2 LIMITATIONS

Considering that more than 85% of indirect branches are return instructions for SPEC CPU2006, Zeus' dynamic reencryption covers the most critical use cases. Zeus also protects other code pointers referenced by the remaining 15 % of indirect branches by static encryption. These code pointers store the function addresses that function designators denote or the GOT has. Conversion of function designators in source programs to moving targets referenced in a chain of copy or arithmetic operations is not as straightforward as converting return addresses. The difficulty comes from tracking code pointers in C and C++ programs where type casting and aliasing occur unrestrictedly. Presently, we are working for dynamic reencryption of the GOTs in the similar framework for return addresses, but we have no immediate plan for extending dynamic reencryption for function designators.

VI. CONCLUSIONS

Compilers and toolchains incorporating Zeus generate executable binaries with the capability of encrypting code pointers. A noteworthy achievement is the function pointer protection. Instead of encrypting or decrypting each function pointer at every reference, Zeus encrypts the function address that can move into the function pointer. Programs preserve encrypted function addresses until indirect branches reference the pointers. Zeus is the first manifestation that has overcome the technical difficulty of encrypting the function pointers.

The most outstanding feature of Zeus is the dynamic reencryption of return addresses. Dynamic

reencryption is a moving target defense that makes executables resistant to leaks from code pointers, and thus, blocks further progress into control-flow interceptions. Zeus-compiled binaries refresh encryption states of return address whenever its execution proceeds into vulnerable regions.

The LLVM compiler infrastructure with Zeus proves the effectiveness in protecting code pointers with acceptable performance overheads. The testing results show that recent attacks with entropy reduction or crash-resistance cannot succeed in disclosing keys and encrypted return addresses and diverting control-flow. The current implementation increases execution time by approximately 2 % to 9 % for C programs, depending on whether the programs are CPU- or IO-bound.

Zeus can substitute for expensive mitigations, such as bound-checking while lowering execution time overheads. Since Zeus-compiled programs are efficient and robust, the compiler and toolchains are appropriate for programming small IoT devices with security.

\

If you have any questions
please contact Alex G. Lee
at alexglee@zeusswdef.com