String Oriented Programming: When ASLR is not Enough

Mathias Payer
Department of Computer Science
ETH Zurich, Switzerland

Thomas R. Gross
Department of Computer Science
ETH Zurich, Switzerland

Abstract

Control-data attacks are a well known attack vector; these attacks either inject new code into running applications or reuse existing code in an unintended way to execute their malicious payload.

Current software systems are protected against control-data attacks using numerous mechanisms like Data Execution Prevention (DEP), stack canaries, and Address Space Layout Randomization (ASLR). ASLR turns deterministic attacks into probabilistic attacks and reduces the probability of a successful attack. Unfortunately, the current ASLR implementation for Linux leaves some memory regions non-randomized. These static memory regions can be used to exploit applications that have ASLR, DEP, and stack canaries enabled.

Format string exploits are an often overlooked attack vector that enables attacker-controlled memory writes in an application. A format string bug exists if a user-supplied string is passed as a first argument to any *printf* function. The only prerequisite for a successful format string exploit is that the attacker must be able to control that format string.

This paper presents String Oriented Programming (SOP), an approach that exploits static memory regions in ASLR enabled applications. SOP uses a format string bug to exploit applications that are protected by a combination of weak ASLR, DEP, and stack canaries. Similar to return oriented programming or jump oriented programming, SOP does not rely on existing code but concatenates gadgets in the application using static program analysis.

1. Introduction

Software running on current computer systems is protected against a wide variety of control-data attacks using three protection mechanisms. First, code injection is no longer possible due to Data Execution Prevention (DEP), which prohibits the CPU from executing instructions on non-code memory pages. DEP uses the executable bit for memory pages in modern CPUs to enable non-executable data regions (i.e., the stack and the heap of the application). A stronger guarantee is $W \oplus X$ which ensures that a page is either writable or executable but not both. Linux uses an $W \oplus X$ approach called *Exec Shield* [26]. Second, canaries [15] (variables with special values) are placed next to buffers on the stack or on the heap and their values are validated to protect the application from buffer overflows. Third, Address Space Layout Randomization [4, 5, 18] (ASLR) is a probabilistic protection that randomizes the locations of code, stack, heap, and other data.

On the other hand, Return Oriented Programming [20] (ROP) and Jump Oriented Programming [7] (JOP) are two modern attack techniques that no longer rely on injected executable code but reuse existing application code. ROP uses so-called stack invocation frames to combine different already existing code snippets. In addition, ROP relies on an unchecked application stack: return addresses must not be verified. Modern runtime guards (e.g., libdetox [19]) use a separate shadow stack to check return addresses and therefore prohibit ROP based attacks. JOP based attacks generalize ROP attacks by exploiting any attacker-controlled indirect control flow instruction and are more complicated to protect against. A runtime system either checks the integrity of every dynamic control flow instruction or the compiler ensures that no open dynamic control flow instructions (e.g., `jmp *%eax`; an indirect jump through the `eax` register) are available in the compiled source (e.g., CFI [1] or XFI [11]).

Both ROP and JOP rely on well-known code locations to be effective (i.e., either on non-randomized code regions or on information leaks in the application that disclose code locations); in addition ROP relies on constant, well-known stack addresses and JOP relies on constant, well-known heap data locations.

Unfortunately, the current ASLR implementation for Linux [18] randomizes only the locations of the stack, the heap, and the code and data locations of all dynamically loaded libraries; the main executable itself is often not randomized. Most applications on Linux are not compiled as

PPREW '13 Jan 26, 2013 Rome, Italy
Copyright © 2013 ACM 978-1-4503-1857-0/13/01. . . $15.00

Position Independent Executables (PIE) and are mapped to a static memory address (currently, only 27 out of all binaries on Ubuntu 11.10 are compiled as PIE). Static code regions create an opportunity for ROP and JOP. The combination of writable sections (e.g., the Global Offset Table - GOT[1]), indirect control flow transfers (e.g., in the Procedure Linkage Table - PLT[2]), and gadgets in the static application can be used to break ASLR. A gadget is a sequence of assembler instructions (not necessarily a function) that already exists in the memory image of the application and executes some specific computation that is valuable to the attacker.

One different class of bugs has not yet received adequate attention in the context of DEP, stack canaries, and ASLR: format string vulnerabilities. If an attacker controls the first parameter to a function of the `printf` family, the string is parsed as a format string. Using such a bug and special format markers result in arbitrary memory writes. Existing exploits use format string vulnerabilities to mount stack or heap-based code injection attacks or to set up return oriented programming. Format string vulnerabilities are not a vulnerability of the past but still pose a significant threat (e.g., CVE-2012-0809 reports a format string bug in `sudo` and allows local privilege escalation; CVE-2012-1152 reports multiple format string bugs in `perl-YAML` and allows remote exploitation, CVE-2012-2369 reports a format string bug in `pidgin-otr` and allows remote exploitation) and usually result in full code execution for the attacker.

This paper assumes the following attack model: an attacker with restricted privileges tries to escalate privileges using String Oriented Programming. The attacker has access to the binary (either because the application is open-source or through some information leak). The attacker is either remote and tries to get user access, or the attacker is local and tries to attack a "SUID" based binary to get administrator privileges. The attacker supplies specially crafted malicious input to the application to exploit a format string bug.

String Oriented Programming (SOP) leverages format string vulnerabilities in combination with ROP and JOP attacks to bypass ASLR, DEP, and stack canaries. We describe an approach that exploits arbitrary non-PIE binaries that use a combination of ASLR, DEP, and stack canaries. Stack canaries are bypassed by using attacker-controlled memory writes that leave the canaries intact. ASLR is circumvented by storing exploit data in well-known locations in static memory regions; control flow is redirected by overwriting GOT slots that are used by code in the PLT. DEP is circumvented by using either ROP or JOP. To add realism to this discussion, this paper focuses on IA32 and Linux but the discussed technique applies to, e.g., Windows and x86-64 as well.

[1] The GOT enables dynamically shared libraries. Every imported function that is used in a module has a corresponding GOT entry. The GOT entry points to the runtime address of the imported function and is resolved by the dynamic loader.

[2] Each runtime module, e.g., a library or an executable, has a PLT that contains function stubs for each imported function. The code in the runtime module calls the local stub in the PLT which then redirects control flow through the GOT to the dynamically resolved imported function.

2. Building blocks for String Oriented Programming (SOP)

SOP relies on an existing format string bug in an application and escalates to any possible control-data attack (code injection) or non-control-data attack (this paper uses either ROP or JOP as examples). This section introduces the different building blocks needed to set up SOP and relates these attack vectors to protection mechanisms that are the security standard of current applications.

2.1 Attack vectors

This section describes existing attack vectors that are used to exploit an application. A successful attack (i) redirects the control flow of the application to an alternate location that would not be reached otherwise (i.e., new code is injected into the application) or (ii) executes already existing code in a different context (i.e., existing code is executed with different - malicious - data). Both forms of attack rely on the following features:

1. The runtime environment must allow the redirection of the control flow to alternate locations using a control flow transfer instruction. Control flow transfer instructions are jump instructions, indirect jump instructions, conditional jump instructions, call instructions, indirect call instructions, return instructions, interrupts, and system calls. Direct control flow transfers encode the target in the instruction and cannot be used for exploits.

   Indirect control flow transfers (indirect jumps, indirect calls, and return instructions) read the absolute target address from a data region or register. An attacker can redirect a legitimate indirect control flow transfer by controlling either the memory location or the specified register (depending on the encoding of the indirect control flow transfer). Exploits either overwrite the register with an attacker supplied value or overwrite the data region that contains the target pointer to achieve the initial control flow redirection: (i) for return instructions the `EIP` on the stack is overwritten, (ii) for indirect calls either a function pointer on the heap, a `GOT` entry in a shared library, or `vtable` entries of objects is changed, or (iii) for indirect jump instructions data-structures of the memory allocator [6] are changed.

2. The exploit must inject some form of payload into the application. Control-data attacks inject machine code instructions into an executable memory region of the application. These instructions are executed after the initial control flow redirection. Data-based attacks like ROP or

JOP modify data structures of the application, a shared library, or the standard loader to execute their malicious payload.

An exploit is only successful if both requirements are met. The following sections present four possible attack vectors in more detail.

2.1.1 Code injection

A code injection attack writes code to an executable region of the application's memory image and transfers control to that injected code [2]. Code injection attacks often use a buffer overflow (e.g., for a C based string or array) to inject code and to overwrite the stored return instruction pointer on the stack in one step. Code injection attacks are *no longer effective* due to increased protection through ASLR, DEP [26], and non-executable stacks that are enabled by default on modern systems.

Code injection is a mature attack vector that has been used for many years. Until recently Intel IA32 did not support the separation of code and data and the CPU tried to interpret any memory region as executable, enabling code injection attacks into data regions. x64, the 64 bit extension of the x86 ISA introduced an additional executable flag for each memory page. The executable flag enables separation of data and code. Only code on pages that have the executable flag set is executed by the CPU. If an exploit redirects control flow to a data page then an exception is triggered. Current operating systems support some form of $W \oplus X$; a memory page is either writable or executable. Due to $W \oplus X$ this attack vector is only applicable if the application uses memory pages that are both writable and executable which is not common except for (i) some old applications that use executable trampolines on the stack, (ii) misconfigured memory regions, or (iii) memory pages that are used to place code that is dynamically generated by a just-in-time compiler.

2.1.2 Return Oriented Programming

Return to libc and Return Oriented Programming [16, 20, 22] (ROP) rely on control of both the stack pointer and the contents of a data buffer on the stack. One way to satisfy these preconditions is through a stack-based buffer overflow. A ROP attack constructs a set of stack invocation frames that are popped one after the other. Each stack invocation frame prepares a set of parameters on the stack and targets a gadget that uses the parameters and executes some computation. A ret2libc attack [23] is a simple ROP attack that uses only one stack invocation frame to execute a libc function (e.g., `system()`) with a set of attacker-controlled parameters. ROP works around DEP but relies on static addresses for the stack and for the gadgets. In addition ROP needs a way to initially redirect control flow to the first ROP invocation frame.

```c
int foo(char *cmp) {
  // assert(strlen(cmp) < MAX_LEN)
  char tmp[MAX_LEN];
  strcpy(tmp, cmp); // missing bound check
  return strcmp(tmp, "foobar");
}
...
// user_str is > MAX_LEN
if (is_foobar(user_str))
...
```

Listing 1. A potential stack-based overflow.

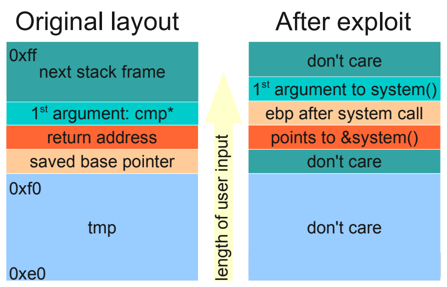

Figure 1. Stack before and after a ROP attack.

Q [22] is a system that analyzes binaries for ROP possibilities. The automatic scanning works around ASLR for non-PIE binaries but relies on a system without stack canaries. Q is therefore not effective on Linux systems like Ubuntu that enable ProPolice (stack canaries) by default.

Listing 1 shows a vulnerable C snippet that is prone to a stack-based buffer overflow. An attacker can inject any data into the buffer and write over the bounds of the buffer to overwrite data structures that are higher up in the stack frame. Figure 1 illustrates a simple return oriented programming attack that exploits the missing bound check in Listing 1 to create one stack invocation frame that executes the `system()` function with forged parameters.

2.1.3 Jump Oriented Programming

Jump Oriented Programming [7, 20] (JOP) is similar to ROP in that JOP manipulates the control flow of the application. Jump oriented data is not limited to stack overflows but uses modified indirect control flow transfers to construct the chain of executed gadgets. Indirect control flow transfers are used in the application to support, e.g., library calls, function pointers (callbacks), and object oriented programming. JOP has similar limitations like ROP and needs static addresses for the gadgets and a known heap location for the JOP dispatcher. In addition JOP needs to redirect control flow to the first JOP dispatcher. ASLR severely limits the initial redirection for JOP on current Linux systems.

Figure 2 illustrates a simple JOP dispatcher on a modified heap object. The dispatcher is a special gadget that uses one register to dispatch individual JOP frames. Each JOP frame contains data and a pointer to a code gadget. The gadget returns control flow to the dispatcher after execution.

2.1.4 Format string attacks

A format string attack [13, 14, 17, 21] exploits that an attacker controls the first parameter to a function of the `printf`-family (all functions that accept a format string as a parameter, e.g., `printf`, `fprintf`, `sprintf`, and `vprintf`). The `printf`-family parses the format string argument for control tokens (of the form `%T`) to determine the number of variable parameters that follow. The token determines how the output on the n-th position on the stack is formatted in the string. Many programmers forget to check user-controlled strings for these control tokens and pass the string directly to the function (e.g., `printf(usr_str)`). A safe implementation would use a static parameter to pass a single string (e.g., `printf("%s", usr_str)`).

The malicious format string can use tokens like `%p` to read specific pointers on the stack, and `%s` to read specific stack addresses as strings. An attacker uses these parameters to get information about the application during the construction of the format string attack.

The `%n` token reverses the order of input and writes the number of already printed characters to the specified pointer. Any argument on the stack can be used as a target address for `%n`, e.g., `%4$hn` writes 2 bytes to the pointer specified $4*4 = 16$ bytes upwards on the stack. The format string itself can be used to store pointers to specific addresses if it is placed on the stack. The number of written bytes can be controlled with additional parameters (e.g., `printf("%NNc");` prints NN bytes) and increases the counter used for `%n`. For example, `printf("AAAA%1$49391c%6$hn")` writes `0xc0f3` (2 bytes, $0xc0f3 - 4 = 49391$) to $0x41414141$ if the string itself is on the 6th slot up on the stack. In this example an input string of 18 bytes length is used to generate an attacker-controlled 2 byte memory write. Format string attacks write arbitrary values to arbitrary memory locations. These attacker-controlled memory writes are used to, e.g., redirect control flow to injected code.

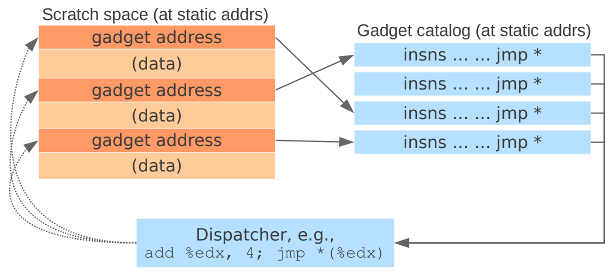

Figure 2. Modified heap object after JOP attack.

2.2 Protection mechanisms

This section discusses protection mechanisms that try to detect possible attacks on different levels of granularity. The protection mechanisms either (i) check the integrity of the stack, (ii) verify library usage, (iii) encrypt pointers, (iv) change the instruction set, (v) protect format strings, (vi) randomize memory locations, or (vii) check and verify every instruction that changes control flow.

Most languages place buffers and variables alongside with return instruction pointers and frame pointers on the regular application stack. Several protection mechanisms [10, 15] verify the stored return instruction pointer on the stack or a canary next to the instruction pointer before the return instruction dereferences the stored address. These mechanisms protect from malicious changes of the return address and the stack layout.

Libsafe/Libverify [3] implements wrappers for library functions that are used in attacks. This approach protects applications from common errors and adds extra checks to "dangerous" functions. A disadvantage of this approach is that it only protects specific functions and general patterns of attack vectors. The glibc has a set of similar patches that are enabled if the `FORTIFY_SOURCE` switch is enabled at compile time on a per-application basis. These patches check every parameter of format strings. Under certain constellations the fortify patches are not secure and can be disabled at runtime [21].

Pointer encryption [9] is an interesting approach to protect instruction pointers from malicious changes. All instruction pointers are encrypted (e.g., using a hash). The application uses the encrypted pointers in all computation (e.g., comparing different function pointers). The compiler adds additional code that resolves the original instruction pointer using the given encrypted pointer whenever it is dereferenced. The attacker does not know the encryption function and therefore cannot forge a pointer to an arbitrary address. Instruction set randomization [12] is a similar approach. The application uses a randomized instruction set and an attacker is unable to guess the instruction set.

Format Guard [8] warns if format strings and functions of the `printf` family are used with unchecked user input. These guards protect the already existing functions of the libc but do not protect from format string exploits in the application code.

Address Space Layout Randomization (ASLR) [4, 5, 18] randomizes all memory regions of an application (e.g., dynamically loaded libraries, heap, and stack). A potential exploit can no longer rely on constant addresses for, e.g., library routines and gadgets. A drawback of this approach is that the address space for 32bit binaries is small and only a few bits can be randomized which opens the possibility of probabilistic attacks [24].

CFI/XFI [1, 11] uses static binary translation to verify every target of all control flow transfers. A set of targets is associated with every control flow transfer location. The group of targets is identified with a secret number. This number is verified when the control flow transfer is executed. The control flow transfer is allowed only if the target number (in the target code) matches the verification code at the source location.

Libdetox [19] is a dynamic binary translation approach that uses runtime information to construct a control flow graph. This control flow graph is enforced at runtime using dynamic checks that are encoded into the translated code.

2.3 Weak ASLR: static regions

The Linux ASLR implementation [18] on x86 is limited if the application itself is not compiled as a Position Independent Executable (PIE). In particular non-PIE ASLR applications are mapped to a constant address (this includes the `data` section, `bss` section, `code` section, `GOT` section, and `PLT` section). An application can be compiled into a PIE, which can then be loaded at random addresses. Linux distributions like Ubuntu only compile a small set of binaries (27 for Ubuntu 11.10) as PIE due to a high performance penalty on x86 [25]. All other programs are compiled without PIE and remain vulnerable in the presence of weak ASLR.

The application image starts at a specific constant address (`0x0804800`). The PLT and GOT regions remain constant as well. The GOT region is writable; writes to a GOT slot can be used to store attack data at static addresses or to redirect control flow by redirecting PLT slots to other locations.

3. String Oriented Programming

String Oriented Programming (SOP) combines a format string vulnerability and ROP/JOP code-reuse techniques into one technique that bypasses ASLR, stack canaries and DEP for non-PIE binaries. SOP uses gadgets that are available in the code region of the application. SOP assumes that some form of DEP is enforced by the system. Otherwise it would be simple to inject some code and to redirect the control flow to the injected code.

Format string exploits enable attacker-controlled writes to attacker-controlled addresses (encoded in the format string) to any memory location that is referenced through a pointer on the stack. An attacker can place arbitrary pointers in an attacker-controlled buffer on the stack; these pointers are then used in the format string attack to write arbitrary memory locations. We assume that the `FORTIFY_SOURCE` patches of the glibc are not enabled (or that we can call `printf` directly). Under some assumptions the fortify patches can be disabled using the format string attack itself or some other auxiliary attack [21]. The fortify patches only add some complexity (and length to the format string).

3.1 Executing code

SOP uses two scenarios to get control of the application without executing injected code. The scenarios are similar to either ROP or JOP. The first (simpler) scenario exploits DEP and stack canaries, while the second scenario exploits DEP, stack canaries, and weak ASLR.

3.1.1 Direct control flow redirect

The first code execution scenario uses an attacker-controlled memory write and a user-controlled buffer on the heap or on the stack (often the format string itself) to prepare the attack. The attacker-controlled memory write redirects control flow by overwriting the return instruction pointer on the stack to a gadget that adjusts the stack frame to the attacker-controlled buffer. The buffer contains a set of invocation frames that concatenate several available gadgets to execute arbitrary code. If the buffer is on the stack then SOP can use ROP, if the buffer is on the heap then SOP can use JOP. This approach combines format string exploits and ROP similar to Section 7.2 of Nergal's paper on advanced return to libc attacks [16].

3.1.2 Indirect control flow redirect

The second code execution scenario uses static data regions in the main application to store all exploit data. Control flow is redirected in two steps: SOP first overwrites the GOT slot that is used by the PLT jump that resolves the next imported function (in the control flow), the application then continues until the next imported function is called. The PLT jump then redirects the control flow to either a ROP gadget or to a JOP dispatcher.

3.2 Resolving addresses

Without ASLR, SOP can be used to easily construct a set of invocation stack frames using direct control flow redirection as in Section 3.1.1 that execute arbitrary functions of the libc (e.g., `mmap` to map an executable memory region, `strcpy` to copy the shellcode), and an indirect control flow transfer to the injected shellcode). If (weak) ASLR is enabled, then the only option for an exploit is to use an indirect control flow redirection as in Section 3.1.2.

Exploits are limited to imported library functions and code sequences available in the application. The dynamic loader resolves the dynamic locations for all imported library functions when they are called, enabling indirect calls of library functions through the `PLT` slots of the application.

The imported library functions in the `PLT` region can be used to resolve unimported and unreferenced functions. The location of resolved library functions are stored at static addresses in the `GOT` region. Gadgets can read and modify these addresses. If the binary of the library is known then the (dynamic) addresses of other library functions that are not imported can be computed. The gadget adds the offset

```
void foo(char *arg) {
  char text[1024];
  if (strlen(arg) >= 1024) return;
  strcpy(text, arg);
  printf(text);
  puts("logged_in");
}
...
foo(user_str);
...
```

Listing 2. A potential format string attack

between the imported function and the requested function to the resolved address in the PLT slot. This enables gadgets to resolve any (unimported) library function whenever a symbol of that library is used.

4. SOP case study

The SOP case study shows how the sample program in Listing 2 can be exploited under different environmental configurations. The host system uses a set of different protection features that are either enabled or disabled. The protection features are ASLR, DEP, stack canaries[15] (protecting the application against buffer overflows), and if specific libc functions are already imported in the main application. The application is compiled without PIE, i.e., the main application image is mapped to a fixed address. This is the standard configuration for Ubuntu 11.10.

4.1 Mo DEP, no ProPolice, no ASLR

If neither ASLR nor DEP are active then the format string contains an attacker-controlled memory write to, e.g., the return instruction pointer, a GOT slot, or a function pointer to redirect control flow to the injected code on the buffer on the stack. This attack conforms to the simple code injection attack in Section 2.1.1.

4.2 DEP, no ProPolice, no ASLR

Exploits rely on data oriented attacks if DEP is enabled. In this configuration a buffer overflow is the simplest solution to set up a ROP attack as described in Section 2.1.2 or a JOP attack as shown in Section 2.1.3.

4.3 DEP, ProPolice, no ASLR

Enabling the ProPolice extension changes the threat landscape and buffer overflows can no longer (easily) be used to exploit systems. ProPolice is enabled by default in recent versions of *gcc*[3]. On the other hand ASLR is not enabled in this configuration therefore a format string attack is used to redirect control flow (by overwriting the return instruction pointer of the `printf` function itself) to a gadget that adjusts

[3] ProPolice is on by default and can be disabled on request using the `-fno-stack-protector` compile-time switch.

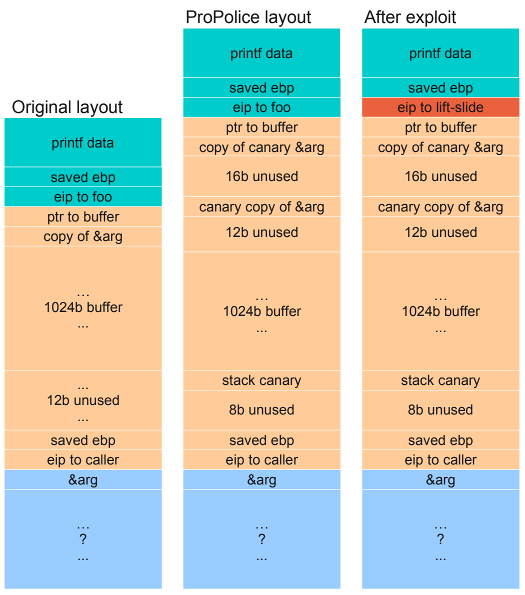

Figure 3. Stack before and after a format string based exploit that prepares stack invocation frames and works around ProPolice (blue: callee, orange: foo, green: printf).

the `%esp` pointer into the user-controlled string. The combination of gcc version 4.5.2 and libc-2.13 adds the function `__libc_csu_init` to all compiled binaries; this function contains a gadget (`add $0x1c,%esp; pop %ebx; pop %esi; pop %edi; pop %ebp; ret`) that lifts the `%esp` by 44 bytes. The format string is prepared so that it contains a set of invocation frames that enable ROP at that specific address.

Figure 3 compares the stack layouts of Listing 2 when the control flow is inside the `printf` function. ProPolice (i) adds a secure canary behind the buffer (that is checked before the return instruction pointer is dereferenced) and (ii) copies the arguments below the buffer. The stack invocation frames in the buffer contain sets of arguments plus return instruction pointers to libc functions (e.g., a call to system would be encoded as `&system; pointer to argument string`). Pointers to, e.g., string arguments, can be directly encoded because the stack addresses are known due to the missing ASLR protection.

4.4 DEP, ProPolice, ASLR, imports available

If ASLR is enabled then the stack and library addresses are no longer constant and therefore unknown. The application itself is located at a static address. In this section we assume that all the gadgets and all the library functions that we want to call are imported in the main application. SOP uses the same basic technique as in Section 4.3 with two differences.

The first difference is that the functions are not called directly but through their `PLT` slots in the application (i.e., the address of `function` is replaced by the address of

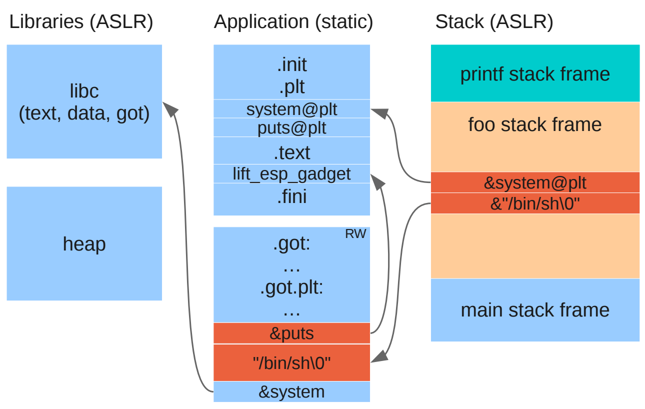

Figure 4. Static and dynamic data regions using weak ASLR. Listing 2 is stopped after the call to `printf`.

`function@PLT`). The PLT section contains an entry for each imported function. This code stub executes an indirect jump through a corresponding GOT entry. The standard loader resolves the correct address for the application. SOP piggy-backs on the standard loading process to resolve the correct references to randomized library functions before they are executed.

The second difference is that addresses on the stack are no longer encoded directly. Any data that is used in the exploit must first be copied to a static region in the application. SOP uses a sequence of format string writes (or an invocation of `strcpy`) to prepare the static data. The invocation frames then use the well-known locations as arguments.

Figure 4 shows a simple exploit that uses a sequence of three four byte writes to (i) write the 8 bytes string "`/bin/sh\0`" into two GOT slots, (ii) overwrite the GOT entry of the imported function that is called next with the lift esp gadget that we used in Section 4.3, and (iii) prepare the beginning of the exploit string with a stack invocation frame that calls `system` trough the static address of `system@PLT`. After `printf` returns the function would continue and try to execute `puts`, the next imported function in our program. But the PLT entry of `puts` is redirected to our *lift esp* gadget that pops some values from the stack and returns into the beginning of the format string on the stack that contains the ROP stack invocation frames. The ROP invocation frame then executes `system("/bin/bash");` (or any other attacker-controlled payload).

4.5 DEP, ProPolice, ASLR, no imports available

If all protection mechanisms are enabled and the application does not import specific library functions then missing function addresses are resolved on the fly. Any missing imports are resolved using the approach described in Section 3.2. A single imported `libc` function allows an exploit to call any sequence of `libc` functions with arbitrary arguments through offset calculation and GOT updating between the different function calls, thereby circumventing ASLR, DEP, and ProPolice.

5. Possible mitigation techniques

This section discusses two possible mitigation techniques to protect applications from SOP. The first mitigation technique discusses a possible extension of ASLR to all memory segments of an application while the second mitigation technique focuses on fixing the `printf` function.

5.1 Full ASLR

Full ASLR randomizes the address layout of the main application (including writable sections like the GOT section) as well as the layout of all libraries. A PIE application has no static memory addresses during subsequent executions. Possible exploits techniques need additional information leaks in the application that disclose the location of code sequences as a first step.

Current applications on IA32 use weak ASLR that is limited to libraries because of performance reasons. The ASLR implementation uses an additional register as a relative code reference pointer. Registers are sparse on IA32 and the use of one of the few registers leads to a huge performance impact.

This section supports the performance overhead claim and presents an evaluation for the PIE feature of GCC that produces position independent executables. The PIE feature enables ASLR for binaries. The evaluation uses GCC version 4.5.2-8ubuntu on Ubuntu 11.04 with Linux kernel version 2.6.38-15-generic and glibc version 2.13. The evaluation system uses an Intel Core i7 dual core CPU clocked at 3.07 GHz with active SMP and 12GB RAM. The evaluation uses all benchmarks of the SPEC CPU2006 v1.01 benchmark suite that compile using recent GCC versions. The evaluation uses two different compilation settings. The benchmarks are compiled with either `-O3` or `-O2`. The benchmarks are executed using the `runspec` program and the configuration uses 3 runs.

Table 1 compares SPEC CPU2006 performance for `-O3` (the most aggressive optimization level of GCC) with and without `-fPIE`. We see that PIE executables are never faster than non-PIE executables and the overhead varies between 0.37% and 26% depending on the benchmark. The benchmarks can be grouped into 4 groups: negligible overhead between 0% and 2% (4 benchmarks), small overhead up to 5% (3 benchmarks), medium overhead between 5% and 10% (5 benchmarks), and high overhead with more than 10% performance penalty (7 benchmarks).

The benchmarks with high overhead either have a highly irregular workload with a large amount of indirect control flow transfers (400.perlbench and 458.sjeng) or process streams of data (401.bzip2, 453.povray, and 482.sphinx3).

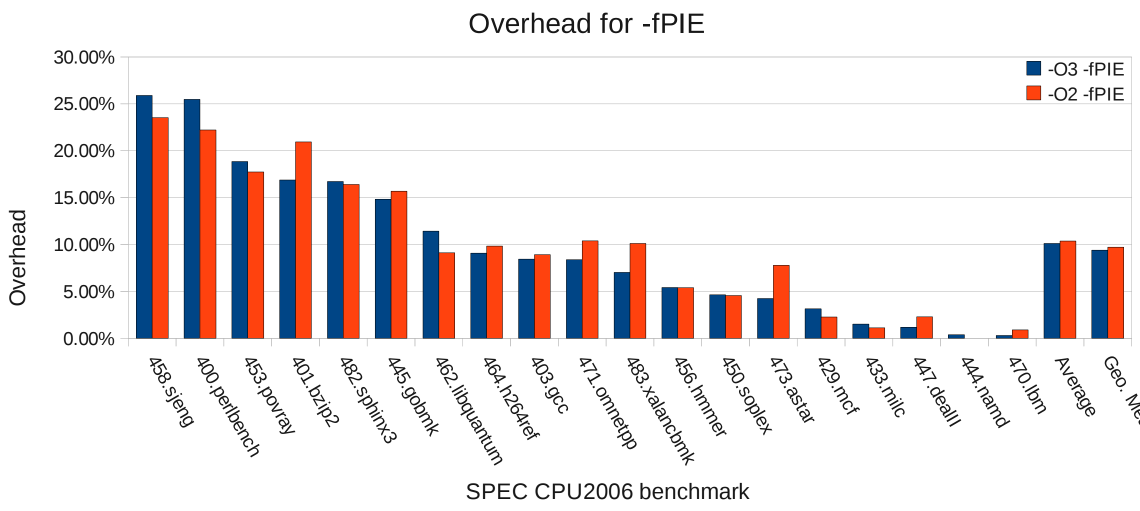

Figure 5. Comparison of the overhead for PIE between -O2 and -O3.

Benchmark	-O3 [s]	-O3 -fPIE [s]	Ovhd. [%]
400.perlbench	369	463	25.47%
401.bzip2	610	713	16.89%
403.gcc	308	334	8.44%
429.mcf	254	262	3.15%
445.gobmk	479	550	14.82%
456.hmmer	554	584	5.42%
458.sjeng	533	671	25.89%
462.libquantum	560	624	11.43%
464.h264ref	760	829	9.08%
471.omnetpp	298	323	8.39%
473.astar	472	492	4.24%
483.xalancbmk	242	259	7.02%
433.milc	394	400	1.52%
444.namd	536	538	0.37%
447.dealII	426	431	1.17%
450.soplex	258	270	4.65%
453.povray	244	290	18.85%
470.lbm	327	328	0.31%
482.sphinx3	520	607	16.73%
Average	429	472	10.12%
Geo. mean	405	443	9.40%

Table 1. Performance of SPEC CPU2006 for -O3 and relative overhead for PIE.

These workloads have a high register pressure and the reduced set of registers is the source for the high overhead.

Figure 5 compares the SPEC CPU2006 results for -O2 and -O3. The benchmarks are ordered by descending overhead for -O3. The overhead for both -O2 and -O3 is comparable for all benchmarks.

The average overhead for PIE (when compiled with O3) is 10% and the geometric mean is 9.4%. This overall non-negligible overhead is the reason why not all applications are compiled with PIE. The Ubuntu distribution chooses performance over the increased security benefit that PIE offers.

5.1.1 Fixing libc

A second possible mitigation strategy is to fix (and patch) the `printf` functions in libc. Either the `%n` token could be removed, or the compiler could ensure that `printf` may only access passed parameters. FormatGuard [8] already presents similar mitigation strategies. Recent GCC versions execute some static checks and print a warning if format strings can be supplied by an external user. In addition libc uses `FORTIFY_SOURCE` to execute additional runtime checks for format strings (e.g., a format string that uses `%n` must be in a non-writable memory area, and if direct parameter access is used in a format string to access the n-th parameter on the stack then all other parameters on the stack between the first parameter and the n-th must be accessed as well in the same string). Due to implementation limitations of libc (that must support both modes with and without `FORTIFY_SOURCE` in a single binary) these patches can be disabled at runtime as described in, e.g., [21].

Even if the libc is fixed and all possible format string vulnerabilities are removed then the problem of the unrandomized application memory image remains. Some static sections of the application are writable on IA32 and can be used to setup an exploit using a data attack that enables a set of attacker-controlled writes to arbitrary memory locations.

6. Conclusion

Current Linux distributions use a variety of security features to protect the running system from security critical bugs in applications. Security features like Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), and stack protection techniques are enabled by default and protect from a variety of attack vectors.

Current systems do not enable Position Independent Executables (PIE) by default; resulting in some static memory regions. String Oriented Programming (SOP) is a possible approach that exploits the weakness of non-PIE applications. SOP uses format string bugs to copy exploit data to static data regions and escalates to Return Oriented Programming or Jump Oriented Programming, thereby effectively bypassing ASLR, DEP, and stack canaries.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *CCS'05: Proc. 12th Conf. Computer and Communications Security* (2005), pp. 340–353.

[2] ALEPH1. Smashing the stack for fun and profit. *Phrack 7*, 49 (Nov. 1996), `http://phrack.com/issues.html?issue=49&id=14`.

[3] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent runtime defense against stack smashing attacks. In *Proc. USENIX ATC* (2000), pp. 251–262.

[4] BHATKAR, E., DUVARNEY, D. C., AND SEKAR, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *SSYM'03: Proc. 12th USENIX Security Symp.* (2003), pp. 105–120.

[5] BHATKAR, S., BHATKAR, E., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *SSYM'05: Proc. 14th USENIX Security Symp.* (2005), pp. 255–270.

[6] BLACKNGEL. The house of lore: Reloaded. *Phrack 14*, 67 (Nov. 2010), http://phrack.com/issues.html?issue=67&id=8.

[7] BLETSCH, T., JIANG, X., FREEH, V. W., AND LIANG, Z. Jump-oriented programming: a new class of code-reuse attack. In *ASIACCS'11: Proc. 6th ACM Symp. on Information, Computer and Communications Security* (2011), pp. 30–40.

[8] COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., AND LOKIER, J. Formatguard: automatic protection from printf format string vulnerabilities. In *SSYM'01: Proc. 10th USENIX Security Symp.* (2001).

[9] COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proc. 12th USENIX Security Symp.* (2003).

[10] COWAN, C., PU, C., MAIER, D., HINTONY, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proc. 7th USENIX Security Symp.* (1998).

[11] ERLINGSSON, Ú., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. In *OSDI'06* (2006), pp. 75–88.

[12] GADALETA, F., YOUNAN, Y., JACOBS, B., JOOSEN, W., DE NEVE, E., AND BEOSIER, N. Instruction-level countermeasures against stack-based buffer overflow attacks. In *VDTS '09: Proceedings of the 1st EuroSys Workshop on Virtualization Technology for Dependable Systems* (2009), ACM, pp. 7–12.

[13] GERA, AND RIQ. Advances in format string exploitation. *Phrack 11*, 59 (2002), `http://phrack.com/issues.html?issue=59&id=7`.

[14] HAAS, P. Advanced format string attacks. `https://www.defcon.org/images/defcon-18/dc-18-presentations/Haas/DEFCON-18-Haas-Adv-Format-String-Attacks.pdf`, DEFCON 18 2010.

[15] HIROAKI, E., AND KUNIKAZU, Y. ProPolice: Improved stack-smashing attack detection. *IPSJ SIG Notes* (2001), 181–188.

[16] NERGAL. The advanced return-into-lib(c) exploits. *Phrack 11*, 58 (Nov. 2007), `http://phrack.com/issues.html?issue=67&id=8`.

[17] OWASP. Definition of format string attacks. `https://www.owasp.org/index.php/Format_string_attack`.

[18] PAX-TEAM. PaX ASLR (Address Space Layout Randomization). `http://pax.grsecurity.net/docs/aslr.txt`, 2003.

[19] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proc. 7th Int'l Conf. Virtual Execution Environments* (2011), pp. 157–168.

[20] PINCUS, J., AND BAKER, B. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy 2* (2004), 20–27.

[21] PLANET, C. A eulogy for format strings. *Phrack 14*, 67 (2010), `http://phrack.com/issues.html?issue=67&id=8`.

[22] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium* (2011).

[23] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07: Proc. 14th Conf. on Computer and Communications Security* (2007), pp. 552–561.

[24] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *CCS'04: Proc. 11th Conf. Computer and Communications Security* (2004), pp. 298–307.

[25] UBUNTU. List of programs built with PIE. `https://wiki.ubuntu.com/Security/Features#pie`, May 2012.

[26] VAN DE VEN, A., AND MOLNAR, I. Exec shield. `https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, 2004.