Computer Security and the Internet: Tools and Jewels

Chapter 8: Software Security—Exploits and Privilege Escalation

P.C. van Oorschot

Aug 30, 2018

Comments, corrections, and suggestions for improvements are welcome and appreciated.
Please send by email to: paulv@scs.carleton.ca

Chapter 8

Software Security—Exploits and Privilege Escalation

This chapter explores basic methods by which common software implementation flaws, design flaws, or design architectures, are exploited by malicious software to gain a foothold on a computer system and/or to "escalate" privileges of a user-space process to that of a root process. Many of these methods abuse memory management systems, such as writing past the end of fixed-length buffers. The methods discussed in this chapter, typically under the heading "software security", are distinct from those in other chapters—e.g., password-guessing attacks (Chapter 3), race conditions (Chapter 5), and web-related injection attacks (Chapter 7). Chapter 9 has greater focus on specific categories of malicious software with classification based on their spreading tactics or end-goals after having gained a foothold—rather than specific technical means by which the foothold is gained.

8.1 Stack-based buffer overflows

BUFFER OVERFLOW. When more water is poured into a glass than it can hold, water spills out. Likewise, the term *buffer overflow* refers to a variable (typically a buffer) having more data bytes written into it than was allocated, thus over-writing whatever data structures or content lies in immediately adjacent memory. Of course, no "proper" memory management system should allow this, but buffer overflows remain a common problem on many computing platforms, for historical and other reasons. When a buffer overflow occurs due to "normal" (non-malicious) system use, the results are unpredictable—the system may crash, program output may be wrong (sometimes unnoticed), or perhaps there is no ill effect at all (e.g., the memory over-written is un-used, or irrelevant to proper program execution). Of greater interest is when an overflow is triggered with malicious intent—in that case, it is called a *buffer overflow attack*, although amongst security experts, the word "attack" is often implied even when not present.

MEMORY LAYOUT (REVIEW). We will use the common memory management layout of Figure 8.1 to explain basic concepts of memory management exploits involving

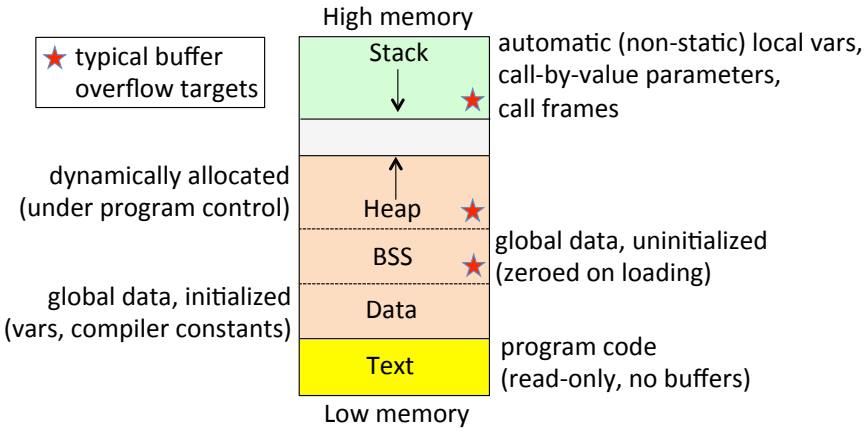

Figure 8.1: Common memory layout for user-space processes. The *data segment* (BSS + Data) part of the heap, containing statically-allocated variables, strings, and arrays, typically grows upwards and is re-organized by calls to memory management functions. BSS (*block started by symbol*) may also be called the block storage segment. Unix-system command line arguments and environment variables are allocated above the stack.

buffer overflows. Principles learned from using this long-standing layout as a baseline remain of use for other layouts, presently and in the future.

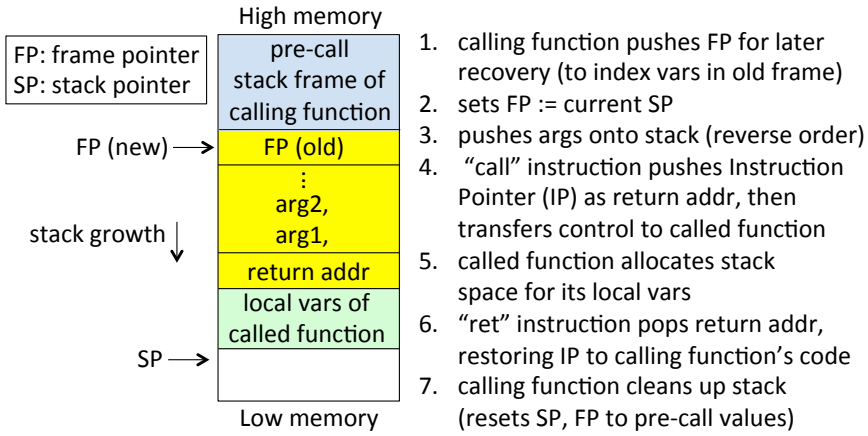

Figure 8.2: User-space stack and function call sequence (C declaration conventions). Frame Pointer (FP) may be called Base Pointer (BP). Saved register state is often also stored on the stack (not shown). Reverse-order arguments facilitate optional parameters.

STACK-BASED BUFFER OVERFLOW. The most well-known form of buffer overflow attack involves variables whose memory is allocated from the stack (Figure 8.1). This is

then called a *stack-based buffer overflow*, or colloquially, *smashing the stack*. Figure 8.2 shows how stack frames are built for individual function calls. Note that local variables (other than those which are assigned to hardware registers) are allocated on the stack. Consider now a local variable that is a buffer, such as `var3` in Figure 8.3. Suppose a program solicits input and writes the input into `var3`, without checking the bounds that limit the size of this buffer; and suppose also that the input is intentionally crafted so as to exceed this limit, over-writing subsequent memory including the stack return address in the calling function's stack frame. When the called function returns, the instruction pointer (program counter) will be assigned the value that overwrote this return address. This results in a transfer of program control to the address specified by the crafted input. One common tactic is to craft the input read into `var3` such that this input itself has a representation which is meaningful machine instructions, and to have the over-written return address point to this code itself within the malicously-crafted input. The transfer of control then begins to execute the injected code itself.

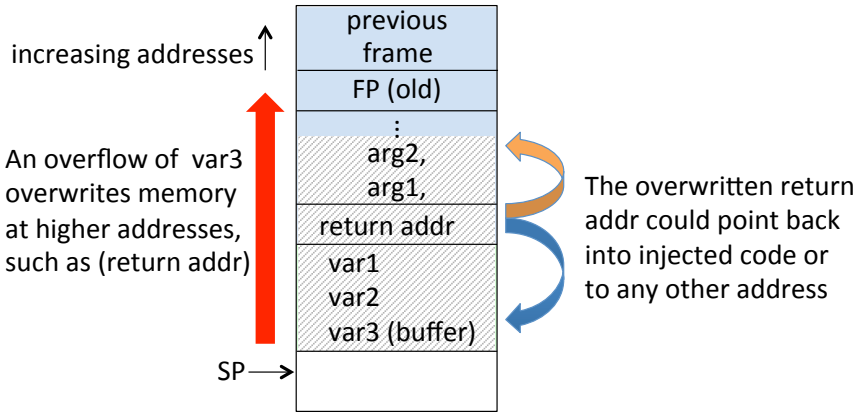

Figure 8.3: Overflow of a local variable on the stack.

‡**NOP SLED.** Crafting injected code for stack execution involves a challenge: precisely predicting the target transfer address that the to-be-executed code will end up at, and within this same injected input, including that target address at a location that will over-write the stack frame's return address. To reduce the precision needed to compute an exact target address, a common tactic is to precede the to-be-executed code by a sequence of machine-code NOP (no-operation) instructions. This is called a *no-op sled*.[1] Transferring control anywhere within the sled results in execution of the code sequence beginning at the end of the sled. Since the presence of a NOP sled is a telltale sign of an attack, attackers may replace literal NOP instructions by equivalent instructions having no effect (e.g., OR 0 to a register). This complicates sled discovery.

[1] This term may make more sense to readers familiar with bobsleds or snow toboggans, which continue sliding down a hill to its bottom (the code to be executed).

8.2 Heap-based buffer overflows

Aside from stack-allocated variables, overflows may affect buffers in heap memory, which includes the data segment (BSS and Data in Figure 8.1). Traditionally, many systems have left the heap and BSS both writable and executable—the latter being unnecessarily dangerous. How dynamic memory is allocated varies across systems (stack allocation is more predictable), but attacker experimentation provides a road map.

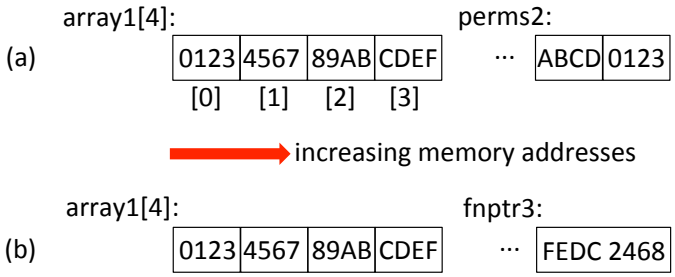

Figure 8.4: Heap overflow. Writing past the end of a heap-allocated buffer can overwrite adjacent heap-allocated variables. Case (a): a permissions-related variable could be over-written. Case (b): a function pointer could be over-written. Both cases highlight the point that program decisions are affected by not only program code itself, but data.

OVERFLOWING HIGHER-ADDRESS VARIABLES. Once an attacker finds an exploitable buffer, and a strategically-useful second variable at a nearby higher memory address, the latter variable can be corrupted. This translates into a tangible attack only if corruption of memory values between the two variables—a typical side effect—does not result in the executing program "crashing" (i.e., terminating due to errors). Figure 8.4 gives two examples. In the first, the corrupted data is some form of permission-related data, e.g., related to or access control; thus, e.g., a FALSE flag might be over-written by a TRUE flag. In the second, what is over-written is a *function pointer*—a variable which holds the address of a function to be called. Over-writing a function pointer is a simple way to transfer control to attacker-selected code, e.g., as an alternative to using the return-address feature used by stack-based attacks for automated control transfer.

GENERAL BUFFER OVERFLOW STEPS. Having considered stack- and heap-based attacks, we see three common steps in exploiting buffer overflows with code injection.

1. *Code injection* (or selection). Code which the attacker aims to be executed is placed somewhere within the target program's address space. If existing system utilities or other code meets the attacker's goal, injection is not needed.

2. *Overflow of a data structure*. One or more data structures are over-flowed, causing corruption to adjacent data. This may be separate from, or part of, the preceding step. The corruption often results in a (subsequent) change in program control flow.

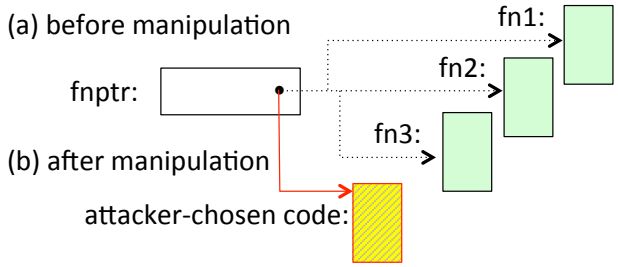

Figure 8.5: Function pointer manipulation. Rewriting a function pointer achieves the objective of changing program control flow. How the attacker-chosen code is selected, or injected into the system, is a separate issue.

3. *Seizure of control*. Program control flow is transferred to the target code (above). This may be by simply waiting after engineering the transfer in the preceding step.

TYPE OF STATE CORRUPTED. For exploits related to memory management, it is instructive to consider the types of variables involved. As noted earlier, program control flow can be directly altered by corrupting data interpreted as a code address, such as:[2]

- stack-based pointers: return addresses, frame pointers;
- function pointers (allocated in the stack, heap or static area), including in tables of function addresses, known as *jump tables*;
- addresses used in C-language `setjmp`/`longjmp` functions. (These are used in non-standard call sequences, such as for exception-handling or co-routines.)

8.3 ‡Return-to-libc exploits

As noted in Section 8.4, buffer overflow attacks that inject code into the runtime stack or heap memory, and then execute that code, can be stopped if support for declaring *non-executable memory* ranges is both hardware-supported and utilized by software. However, this defense does not stop *return-to-libc* attacks, as described next.

STACK-BASED RETURN-TO-LIBC ATTACK. Such an attack may proceed as follows. A return address is over-written as in stack-based attacks above, but now it is pointed to transfer execution not to attack code located on the stack itself, but to existing (authorized) system code, e.g., implementing a system call or a standard library function in libc—with parameters arranged by the attacker. A particularly convenient such function is system(), which takes one string argument, with resulting execution as if the string (typically the name of a program plus invocation parameters) were entered at a shell command line. Unix-type operating systems implement system() by using fork() to create a child process

[2] Flow can be altered indirectly if data corrupted is used in a test for a branching decision. Explain***

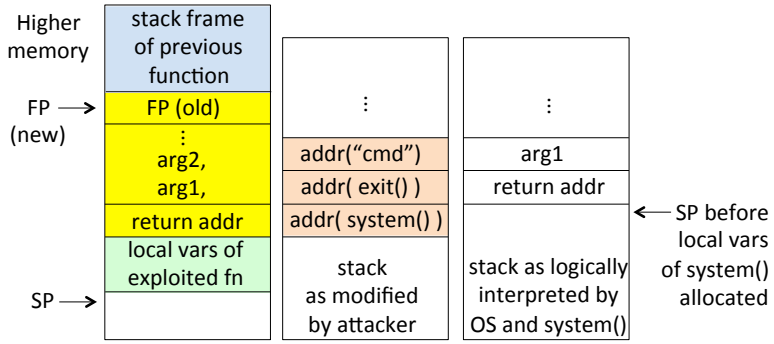

Figure 8.6: Return-to-libc attack on user-space stack. A local stack variable is overflowed such that the return address becomes that of the library call system(), which will expect a "normal" stack frame upon entry, as per the rightmost frame. If what will be used as the return address upon completion of system() is also overwritten with the address of the system call exit(), an orderly return will result rather than a likely memory violation error.

which then executes the command using execl() per Section 8.8, and returning from system() once the command has finished. The call to execl() may be of the form

```
execl("/bin/sh", "sh", "-c", cmd, 0x00)
```

to invoke `bin/sh` (commonly the `bash` shell), instructing it to execute the command `cmd`. The attacker puts the parameter `cmd` in the stack location where the invoked library function would normally expect to find it as an argument. See Figure 8.6.

‡**Exercise** (Exploiting heap-management utilities). C-language programs allocate dynamic memory using malloc(), supported by underlying system calls to utilities that manage heap memory in *chunks*. Commonly, each chunk starts with header fields indicating if the block is free or allocated, and the location or address of the next chunk, e.g., using a singly- or doubly-linked list with pointers. Overflowing a buffer allocated from such heap memory can thus over-write pointers. Outline how this may allow malicious over-writing of arbitrary memory locations. (Hint: [5], also [6, 119-123].)

‡**Exercise** (Format string vulnerabilities). A powerful class of exploits takes advantage of how *format strings* interact with system memory management in function families such as the C-language `printf(%s, string)`. When the format string `%s` is a dynamic variable (rather than static), the rich functionality provided allows reading and writing at arbitrary memory addresses. Look up and explain how format string attacks work; summarize possible defenses (hint: [15] and [6, 125-128]).

8.4 Buffer overflow defenses and adoption barriers

ADOPTION BARRIERS. Many measures have been proposed to stop buffer overflow attacks. We mention a few below, to emphasize both the wide variety of possible counter-

measures, and the difficulty of deploying any one of them on a wide basis. The latter is due to fundamental realities about today's worldwide software ecosystem, including:

- *(no single governing body)* no corporation, country, government, or standards organization has the ability to set and enforce rules on software-based systems worldwide, even if ideal solutions were known;
- *(backwards compatibility)* proposals to change software platforms or tools that introduce interoperability problems with existing software or cause any in-use programs to cease functioning, are immediately rejected; and
- *(incomplete solutions)* proposals addressing only a subset of exploitable vulnerabilities, at non-trivial deployment or performance costs, meet cost-benefit resistance.

Clean-slate approaches that entirely stop exploitable buffer overflows in new software are of interest, but leave use vulnerable to exploitation of widely deployed and still heavily relied upon legacy software. The enormous size of the world's installed base of software, particularly legacy systems written in vulnerable (see below) C, C++ and assembler, makes the idea of modifying or replacing all such software impractical, for several reasons (e.g., cost, lack of available expertise, unacceptability of disrupting critical systems). Nonetheless, much progress has been made, with many approaches now available to mitigate exploitation of memory management vulnerabilities such as buffer overflows.

BUFFER OVERFLOW COUNTERMEAURES. Measures to counter buffer overflow attacks may either prevent (or reduce) occurrences of vulnerabilities in programs (e.g., by compile-time tools that flag vulnerabilities), or prevent the exploitation of vulnerable programs (e.g., through run-time mechanisms, with run-time overhead). Well-known proposals include the following.

1. **NON-EXECUTABLE STACK AND HEAP.** Buffer overflow attacks that execute injected code directly on the stack or heap itself can be stopped if hardware support exists to flag specified address ranges as *non-executable memory*. For example, any address range assigned to the stack or heap can then be marked invalid for loading via the Instruction Pointer (Program Counter). However, for backwards compatibility reasons, this feature may be disabled even if available.

2. **RUNTIME STACK PROTECTION.** *Stack canaries* are check-words used to detect some code injection attacks. An extra field is inserted in stack frames, just below (i.e., at lower address than) an attack target like a return address—in Figure 8.2, just above the local variables. A naive buffer overflow attack that corrupts all memory between the buffer and the return address will over-write such a canary. A runtime system check looks for an expected (canary) value in this field before using the return address. If the canary word is not correct, an error handler is invoked. *Heap canaries* work similarly; any field (memory value) may be protected this way. Related approaches are *shadow stacks* and *pointer protection* (e.g., copying return-addresses to OS-managed data areas then cross-checking for consistency before

use; or encoding pointers by XOR of a secret mask, such that attacks overwriting a pointer may corrupt it but not successfully modify control flow).

3. **RUNTIME BOUNDS-CHECKING.** Here, compilers instrument code to invoke runtime support which tracks, and checks for conformance with, bounds on buffers.

4. **MEMORY LAYOUT RANDOMIZATION.** Code injection attacks often require precise calculations of memory addresses, and typically rely on predictable (known) memory layouts on target platforms. This can be disrupted by defensive approaches which randomize the layout of objects in memory, including the base addresses used for runtime stacks, heaps, and executables including runtime libraries.

5. **TYPE-SAFE LANGUAGES.** C, and similar systems languages that favour efficiency and programming flexibility over safety, allow *type casting* (the ability to convert variables from one data type to another), and unchecked *pointer arithmetic* (e.g., a pointer can be used as the base to index an array, whether or not space was allocated for an array at all, let alone one of the size implied by the index).[3] The combination of these features leads to many buffer overflow vulnerabilities. In contrast, *strongly-typed* or *type-safe languages* (e.g., Java, C#) tightly control data types, and automatically enforce bounds on buffers, including run-time checking. A related alternative is to use so-called safe dialects of C.

6. **SAFE C LIBRARIES.** Another root cause of buffer overflow vulnerabilities in C-family languages is the manner in which character strings are implemented, and the system utilities for string manipulation in the standard C library, libC. As a background reminder for C: character strings are arrays of characters, i.e., buffers; and by efficiency-driven convention, the presence of a null byte (`0x00`) defines the end of a character string. An example of a dangerous libc function is `strcpy(s1, s2)`. It copies string `s2` into string `s1`, but does no bounds-checking. Consequently, various proposals promote use of "safe C library replacements" instead of historical (unsafe) libC functions such as string-handling routines without bounds-checks. One approach is to instrument compiler warnings instructing programmers to use substitute functions; this of course does not patch existing code.

Other approaches to mention:[4]

- compile-time tools (static analysis) to detect memory management vulnerabilities.
- input sanitization (filtering, quoting/escaping input) by applications. Various means check that input follows expectations, to rule out executable content where possible.
- dynamic analysis tools; taint analysis.
- fuzz-testing.

[3] Hand-coded assembly is similarly unrestricted.

[4] ***To be completed

‡**Exercise** (Control flow integrity). Summarize how compile-time (static) analysis can be combined with runtime instrumentation for *control flow integrity*, which stops transfer of program control to addresses inconsistent with compile-time analysis (hint: [1, 2]).

8.5 Privilege escalation and the bigger picture

Once attackers get code of their own choosing to run, a next step is to elevate privileges, moving from a base toehold (e.g., user-space privileges) into a position of higher power (root/kernel-level privileges). For network daemons receiving packets, it may be just a single step into root privileges (e.g. spawning a new shell); daemons may be run as root for convenience, e.g., to bind to low-numbered ports. In other cases, a local-machine user program triggers the access.

One form of privilege escalation is to move from a constrained program to an uncontrained shell from which arbitrary commands (including other programs) can be run. Recall (Chapter 5) that race conditions can also be used to escalate privileges. Another path to full prileges is to arrange a shell be spawned by a fully-privileged program, e.g., a root-owned setuid program. Attack outline (***to be refined***):

1. Assume the attacker has gained local access to a regular user-privilege process on a machine. Locate a root-owned setuid program, e.g., that accepts user (attacker) or network input, with a suitable buffer overflow vulnerability.

2. Craft suitable exploit input, and supply it. If the existing access was through a normal user shell, this may be via command line input.

3. Assume the overflow resulted in a root shell. (If from a command line input, the new shell would be an overlay of the child process forked by the user shell to execute the command.) The attacker could type into this shell if local, via `stdin`, or scripted commands read in from a stored file or retrieved over the network.

***To be completed.

8.6 Integer overflows and other exploits

***Omit for now (to be completed): integer overflow + code injection attacks:

- those covered in Chapter 7 (web security): XSS, CSRF, SQL
- those in this chapter: shellcode injection, OS command-line injection (Section 8.7).

8.7 Environment and configuration-related exploits

***Omit for now (to-be-completed).

Abusing shells, environment variables (e.g., PATH), start-up files.

Software management, installation, update.

8.8 ‡Background: process creation, syscalls, shells, shellcode

Here we review basic concepts that aid in understanding attacker exploits in the chapter. We use Unix examples; other systems operate analogously. Recall that Chapter 5 reviews process creation, fork() and exec(), inheritance of parent UIDs, and setuid programs.

SHELLCODE (TERMINOLOGY). A command shell provides a convenient, known interface to system utilities. From it, arbitrary system commands can be run, or other programs by specifying the program name (filepath if needed) and arguments. Thus a common attack goal is to obtain a shell running with root privileges, e.g., by causing a process to transfer execution to an instruction sequence that creates a new shell process that then cedes execution as explained below. The term *shellcode* is used narrowly to refer to typically short sequences of injected code that create a command shell (ideally, root shell) when executed; this is the literal and historical usage. More broadly, shellcode refers to injected code that when run, achieves a specific attack task—possibly transferring control to a longer stream of attack instructions or launching further (malicious) executables.

SYSCALLS AND C LIBRARY (BACKGROUND). Low-level kernel operations such as reading and writing files depend on details specific to particular operating systems and hardware platforms. These operations are implemented by *syscalls* (system calls), which themselves are often accessed through C-language *wrapper functions* closely resembling each, packaged in a common user-space C library, *libc*. The libc functions make syscalls after handling system-specific details, e.g., using assembly code to load parameters in registers depending on platform conventions, and invoking a TRAP or software interrupt switching the processor mode from user to supervisor/kernel. The syscall implementations, running in kernel model, have access to kernel resources.

COMMAND SHELLS AND FORK (BACKGROUND). An operating system (OS) *command line interpreter* (shell) is not part of the core OS, but provides access to many OS features and is a main system interface for users (graphical user interfaces, or GUIs, are an alternate interface). When a Unix user logs in from a device (logical terminal), the OS starts up a shell program as a user process, waiting to accept commands; the user terminal (keyboard, display) is configured as default input and output channels (`stdin`, `stdout`). When the user issues a command (e.g., by typing a commond at the command prompt, or redirecting input from a file), the shell creates a *child* process to run a program to execute the command, and waits for the program to terminate. This proceeds in Unix by calling fork(), which clones the calling process; the clone recognized as the child then calls execve() to replace its image by the desired program to run the user-requested command (below). When the child-hosted task completes, the shell provides any output to the user, and prompts the user for another command. If the user-entered command is followed by an ampersand (`&`), the forked child process operates in the background and the shell immediately prompts for another command. For an analogous shell on Windows systems, `"cmd.exe"` is executed.

EXECVE CREATION OF A SHELL (BACKGROUND). An example of C code to create an interactive shell is as follows:

```
char *name[2];
   name[0] = "sh";
   name[1] = NULL;
   execve("/bin/sh", name, NULL);
```

We may view execve() as the core exec() family system call, with general form:

`execve(path, argv[ ], envp[ ])`.

Here `path` (pointer to string) is the pathname of the file; `"v"` signals a vector `argv` of pointers to strings (the first of which names the file to execute); `"e"` signals an optional `envp` argument pointer to an array of environment settings (each entry a pointer to a null-terminated string name=value), and NULL pointers terminate `argv` and `envp`. The `exec` family calls launch (execute) the specified executable, replacing the current process image, and ceding control to it (file descriptors and process-id are inherited or passed in). The filename in the `path` argument may be a binary executable or a script started by convention with: `#! interpreter [optional-arg]`. Other `exec` family system calls may be front-ends to execve(), e.g., `execl(path, arg0, ...)` where `"l"` denotes *list*: beyond `path`, individual arguments are in a NULL-ended list of pointers to null-terminated strings; `arg0` specifies the name of the file to execute (usually the same as in `path`, the latter being relied on to locate the executable). Thus alternative code to start up a shell is:

`char *s = "/bin/sh"; execl(s, s, 0x00)`

Compiling this C code results in a relatively short machine code instruction sequence, easily supplied by an attacker as, e.g., program input to a stack-allocated buffer. Note that the kernel's `exec` family syscall then does the bulk of the work to create the shell.

‡SHELLCODE: TECHNICAL CHALLENGES. Some tedious technical conditions constrain binary shellcode—but pose little barrier to diligent attackers, and solutions are easily found online. Two main challenges are: *eliminating NULL-bytes* (`0x00`) within injected code, and *relative addressing*. NULL-bytes affect string handling utilities. Before injected code is executed as shellcode, it is often processed by libc functions—for example, if injection occurs by a solicited input string, then string processing routines will treat a NULL byte in any opcode as end-of-string. This problem is overcome by use of alternate instructions and code sequences avoiding opcodes containing `0x00`. Relative addressing is necessary due to not knowing the physical address at which shellcode will itself reside; this is overcome by code arranging that a machine register is loaded with the address of an anchor shellcode instruction, and then selecting machine operations that use addressing relative to that register. Once one talented individual figures out such details, automated tools often allow easy replicatation for others.

8.9 ‡Endnotes and further reading

REFERENCES AND FURTHER READING. The shell code in Section 8.8 is from Mudge (Peiter Zatko) [14], predating the reader-friendly stack-based buffer overflow road-map of Aleph One (Elias Levy) [3]. Conover [7] offers a tutorial on heap-based buffer overflows.

The teaching on format string vulnerabilities by Scut [15] follows a December 2000 talk hosted by the Chaos Communication Club (`https://events.ccc.de/congress/`).

For background on operating systems, command shells and system calls, see Tanenbaum [16]; and Anley [4], especially as they relate to shellcode.

Aycock [6].

Viega-McGraw 2001 [13].

Hoglund-McGraw 2004 [9].

McGraw 2006 [12].

McGraw-Felten on Java security [13].

End-to-end overflow example with privilege escalation to shell.

Early surveys on buffer overflows.

Return-oriented programming (ROP).

Forrest [8] discusses the idea of randomizing the memory layout of stacks and other data; related *address space layout randomization* (ASLR) techniques were popularized by the Linux PaX project circa 2001. Keromytis [11]) surveys other proposals to counter code injection using randomization, including *instruction set randomization*.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 340–353, 2005. See also journal version (ACM TISSEC 2009).

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symp. Security and Privacy*, pages 263–277, 2008.

[3] Aleph One. Smashing the Stack for Fun and Profit. 8 November 1996, *Phrack* vol.7 no.49, file 14 of 16, `http://www.phrack.org`.

[4] C. Anley, J. Heasman, F. Lindner, and G. Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes (2/e)*. Wiley, 2007. 718 pages. First edition (2004, J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, R. Hassell), 620 pages.

[5] anonymous. Once upon a free()... *Phrack* vol 11 issue 57, file 9 of 18 (11 Aug 2001), `http://www.phrack.org`.

[6] J. Aycock. *Computer Viruses and Malware*. Springer Science+Business Media, New York, NY, 2006.

[7] Conover, Matt & w00w00 Security Development (WSD). w00w00 on Heap Overflows. January 1999, `http://www.w00w00.org/articles.html`.

[8] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *IEEE HotOS*, pages 67–72, 1997.

[9] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.

[10] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM Conf. Computer and Comm. Security (CCS)*, pages 272–280, 2003.

[11] A. D. Keromytis. Randomized instruction sets and runtime environments: Past research and future directions. *IEEE Security & Privacy Magazine*, 7(1):18–25, 2009.

[12] G. McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006. Includes extensive annotated bibliography.

[13] G. McGraw and E. W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes*. John Wiley, 1997. Available as a free web edition; the 2nd edition (1999) has the title *Securing Java*.

[14] mudge. How to write Buffer Overflows. 20 October 1995, available online.

[15] scut / team teso. Exploiting Format String Vulnerabilities (version 1.2). 1 September 2001, available online.

[16] A. S. Tanenbaum. *Modern Operating Systems (third edition)*. Pearson Prentice Hall, 2008.