Chapter 6

Software Security—Exploits and Privilege Escalation

The official version of this book is available at
https://www.springer.com/gp/book/9783030834104

ISBN: 978-3-030-83410-4 (hardcopy), 978-3-030-83411-1 (eBook)

version: 29 June 2021

Chapter 6

Software Security—Exploits and Privilege Escalation

Here we consider common methods that exploit vulnerabilities in (typically non-security) software programs, through abuse of features in programming languages, system architectures, and supporting functionality. *Software security* aims to address these problems, and is distinct from security software. Once malicious software gains a foothold on (entry point into) a computer system, this is often followed by attempts to elevate privileges from those of a regular user to a superuser (Administrator on Windows, or Unix/Linux root, e.g., allowing access to all user files and privileged software commands, including to change permissions of other users), or further to supervisor level. Many software security problems stem from weak memory management controls; the headline example involves exploiting buffer overflows (indexing beyond the bounds of fixed-length buffers).

For context, software security attacks differ from stealing or guessing passwords (Chapter 3), web-related injection attacks (Chapter 9), and Chapter 7's discussion of categories of malware based on spreading tactics or end-goals after having gained initial access. For example, rootkits (Chapter 7) may use buffer overflow exploits and *shellcode* to gain access and then additional techniques to maintain a hidden presence; herein we discuss the technical means to gain the foothold.

This chapter spends most of its pages outlining how attacks work, i.e., illustrating insecurity. Why so? Understanding attacks is necessary in order to devise technical defenses. Attackers are well-versed in their art, widely sharing knowledge and point-and-click toolkits on shady web sites. Should defenders remain uneducated? Presenting concrete attack details in understandable language also helps cut short arguments about whether attacks are feasible. Even better is live demonstration on fielded systems (with prior permission) as in *penetration testing* (Chapter 11). While anti-virus software, firewalls and intrusion detection systems are useful defensive tools, software security has emerged as its own defensive subdiscipline, from the realization that a major root cause of security problems is poor quality of ordinary software itself, independent of the things around it, or security-specific mechanisms within it.

6.1 Race conditions and resolving filenames to resources

As noted in Chapters 1 and 5, principle P4 (COMPLETE-MEDIATION) states that a system should verify authorization before granting access to a resource, ideally immediately prior. The timing concern is due to *races*, discussed here in the limited context of filesystem races. We also consider the broader issue of resolving filenames to the expected resources.

TOCTOU RACE. Suppose an access control file permission check is made at time t_1, and the object is accessed at time $t_2 > t_1$. A common implicit assumption is that the condition checked does not change from *time-of-check to time-of-use* (TOCTOU). But in multi-processing systems with interrupts, things do change—e.g., metadata like file permissions and owners, the object a filename resolves to, or arguments passed to called routines. When a condition check made at one instant is relied on later, and there is a chance that in the interval something changes the outcome—including due to malicious actions that increase the likelihood of change—then an exploitable *race condition* may exist. This situation has occurred in a surprising diversity of situations, and such TOCTOU *races* require special attention by system designers and developers.

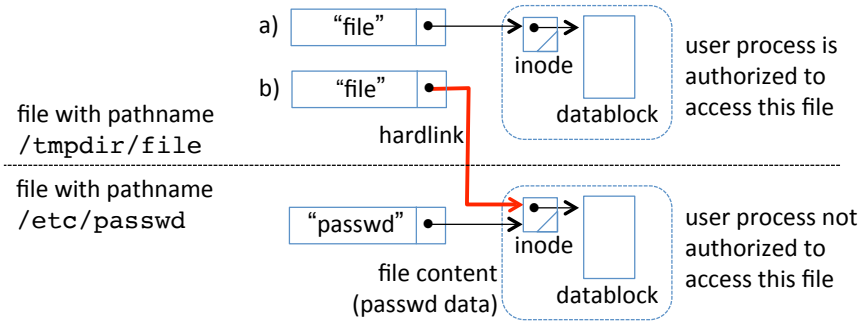


Figure 6.1: Filesystem TOCTOU race. a) A user process U is permitted to write to a file. b) The file is deleted, and a new file created with the same name references a resource that U is not authorized to write to. The security issue is a permission check made in state a) and relied on later, after change b). See Chapter 5 for filesystem structures.

Example *(Privilege escalation via* TOCTOU *race).* As a concrete Unix example, consider P, a root-owned `setuid` program (Chapter 5) whose tasks include writing to a file that the invoking user process U supposedly has write access to. By a historically common coding pattern, P uses the syscall access() to test whether U's permissions suffice, and proceeds only if so; this syscall check uses the process' real UID and GID, e.g., `rUID`, whereas open() itself uses the effective UID (`eUID`, which is `root` as stated, and thus always sufficiently privileged). The access test returns `0` for success (failure is `-1`). Thus P's code sequence is:

```
1: if(access("file",PERMS_REQUESTED)==0)then
2:   filedescr = open("file", PERMS)    /* now proceed to read or write */
```

But after line 1 and before line 2 executes, an attacker alters the binding between the filename and what it resolved to in line 1 (see Fig. 6.1). Essentially, the attacker executes:

```
1.1:  unlink("file")                    /* delete name from filesystem  */
1.2:  link("/etc/passwd", "file")       /* new file entry links to passwd */
```

Now when P proceeds to write to "`file`", it overwrites the password file. Process U does not have permission to do this, but P does (as root-owned and `setuid`). Due to this issue (dating back to 1993), use of access() for such checks is now largely discouraged.

MITIGATING RACES. The underlying problem is that lines 1, 2 do not execute as an atomic pair, and the mapping of filename to referenced object (inode) changes; *atomic* actions are desired, as in transaction systems. Simply disabling interrupts is not a viable solution—not all interrupts can be safely disabled, not all processes can wait while others run uninterrupted, and in multi-processor systems that share memory and other resources, this may require disabling interrupts on all processors. For the example above, one suggested alternative to using access() is to set the eUID to rUID (and similarly eGID to rGID) before calling open(), although this approach is not portable across OSs due to system library inconsistencies, e.g., in setuid(). A second alternative is to drop privileges then fork() an unprivileged child, which proceeds with an open() attempt and, if successful, makes the file descriptor accessible to the parent before exiting; this also tends to be non-portable. A standard means to avoid file access races is wherever possible to use system calls that deal directly with file descriptors (they are not subject to change, whereas the referent of a filename may); however, system calls that support filenames do not always have equivalent support for file descriptors. For other means to mitigate filename-based privilege escalation including TOCTOU races, see the end notes (Section 6.9).

Example *(Safe-to-resolve filenames).* Intuition can mislead. Consider an attempt to make a filename "safe" to use, in the sense of ensuring an authentic file-to-resource resolution, immune to malicious alteration.[1] Directory entry *hopefully* (Fig. 6.2) references a directory file (inode) that user `hope` owns and has exclusive R, W, X permissions on, i.e., no other regular users or groups have any permissions on this inode. Next `hope` creates a regular file *safe* in this controlled directory. Now `hope` appears to have control over two files (inodes), at levels 4 and 5. The question is: If a process running under `hope`'s UID tries to access file *hopefully/safe*, is it guaranteed that the resource accessed will be the one just created? The answer is no. If the full pathname is */usr/zdir/hopefully/safe*, and malicious (non-root) user `tricky` has full permissions on the (level 3 in figure) inode referenced by the *zdir* directory entry, then `tricky` can remove (by renaming) *hopefully* from *zdir*, and create an entirely new directory file, and regular file, both using the old names. Now a file reference to */usr/zdir/hopefully/safe* retrieves `tricky`'s bogus file *safe*. The lesson is that a method aiming to guarantee "safely resolvable" filenames must be wary of parent directories up to "/". Note here that even if `hope` controlled the inode at level 3, an attacker with control of the level 2 inode could cause similar problems.

‡**Example** *(/tmp file exploits).* The following is a typical exploit on world-writable directories, used by utility programs for temporary files. Directory permissions are commonly `1777` (including the sticky bit, Chapter 5). A well-known C compiler (gcc) gener-

[1] The (false) idea is that safety results from controlling write permission on files within directory *hopefully*. Attacks to beware of here are related to principle P19 (REQUEST-RESPONSE-INTEGRITY).

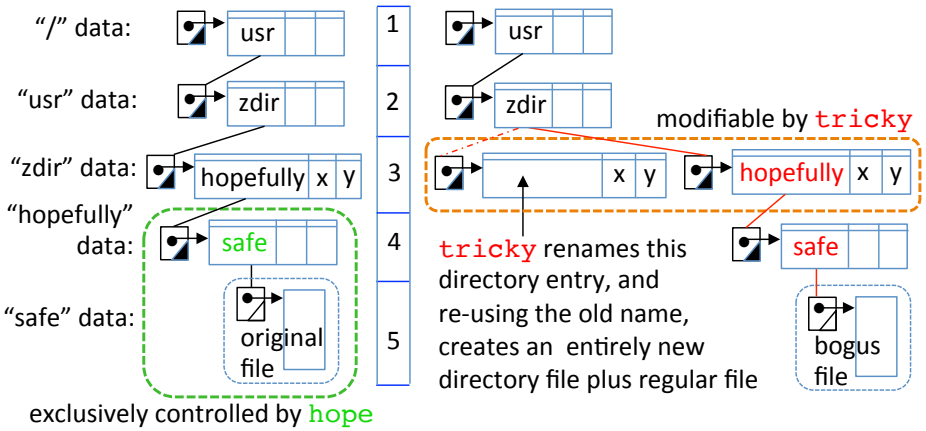


Figure 6.2: An attempt to make a filename "safe to resolve". The directory at level 4 (left) is under the control of user `hope`, i.e., no other regular user has any R, W or X permissions on this level 4 inode. Malicious user `tricky` has full permissions on the level 3 inode. Chapter 5 gives background on inodes and directory structure.

ates a sequence of files, e.g., intermediate (`.i`), assembly (`.s`), and object (`.o`). For these, it uses a unique filename prefix (denoted by `zzzzzz` here) generated as a random string by a system function. Knowing this, an attacker program can await the appearance of new `.i` files, and itself create (before the compiler) a symlink file with name `/tmp/zzzzzz.o`, symbolically linked to another file. If the compiler process has sufficient privileges, that other file will be overwritten in the attempt to access `/tmp/zzzzzz.o`. This may be called a *filename squatting attack*. For certain success, the attack program awaits a root process to run the compiler (root can overwrite any file). A subtle detail is the compiler's use of a system call open() with flag `O_CREAT`, requesting file creation unless a file by that name pre-exists, in which case it is used instead. (Example based on: Dowd [25, p.539].)

‡**Exercise** (TOCTOU race: temporary file creation). An unsafe coding pattern for creating temporary files involves a `stat-open` sequence similar to the first example's `access-open` sequence, and similarly vulnerable to an attack, but now using a symbolic (rather than a hard) link. a) Find a description of this pattern and attack, and provide the C code (hint: [60]). b) Explain the attack using a diagram analogous to Figure 6.1.

6.2 Integer-based vulnerabilities and C-language issues

Integer-based vulnerabilities are exploitable code sequences due to *integer bugs*: errors related to how integers are represented in memory. They arise from arithmetic operations, and from side effects of type conversions between integer types of different widths or *signedness* (signed vs. unsigned). We explain how these vulnerabilities occur, how they can be exploited, and approaches to address the issues. Integer-based vulnerabilities do not themselves involve injection of executable code, shell commands, or scripts, and thus are distinct from buffer overflow vulnerabilities (where a further issue is failure to enforce

address bounds on memory structures) and other attacks involving code injection.

FOCUS ON C. We focus on C, as the biggest problems arise in C-family programming languages (including C++). While some vulnerabilities occur more widely—e.g., integer overflows (below) occur in Java—C faces additional complications due to its eagerness to allow operations between different data types (below). Moreover, security issues in C have wide impact, due to its huge installed base of legacy software from being the historical language of choice for systems programming, including operating systems, network daemons, and interpreters for many other languages. Studying C integer-based vulnerabilities thus remains relevant for current systems, and is important pedagogically for its lessons, and to avoid repeating problematic choices in language design.

C CHAR. To begin, consider the C `char` data type. It uses one byte (8 bits), and can hold one character. It is viewed as a small integer type and as such, is commonly used in arithmetic expressions. A `char` is converted to an `int` (Table 6.1) before any arithmetic operation. There are actually three distinct types: `signed char`, `unsigned char`, and `char`. The C standard leaves it (machine) "implementation dependent" as to whether `char` behaves like the first or the second. A `char` of value `0x80` is read as $+128$ if an unsigned integer, or -128 if a signed integer—a rather important difference. This gives an early warning that operations with C integers can be subtle and error-prone.

Data type	Bit length	Range	
		unsigned	signed
`char`	8	0..255	$-128..127$
`short int`	16	0..65535	$-32768..32767$
`int`	16 or 32	0..UINT_MAX	INT_MIN..INT_MAX
`long int`	32	$0..2^{32}-1$	$-2^{31}..2^{31}-1$
`long long`	64	$0..2^{64}-1$	$-2^{63}..2^{63}-1$

Table 6.1: C integer data types (typical sizes). C integer sizes may vary, to accommodate target machines. Type `int` must be at least 16 bits and no longer than `long`. Undeclared signedness (e.g., `int` vs. `unsigned int`) defaults to signed, except for `char` (which is left machine-dependent). The C99 standard added exact-length signed (two's complement) integer types: `intN_t`, `N` = 8, 16, 32, 64. For an n-bit `int`, UINT_MAX = $2^n - 1$.

INTEGER CONVERSIONS. C has many integer data types (Table 6.1), freely converting and allowing operations between them. This flexibility is alternatively viewed as dangerous looseness, and C is said to have *weak type safety*, or be *weakly typed*. (As another example of weak type safety in C: it has no native string type.) If an arithmetic operation has operands of different types, a common type is first arranged. This is done implicitly by the compiler (e.g., C automatically *promotes* `char` and `short` to `int` before arithmetic operations), or explicitly by the programmer (e.g., the C snippet "`(unsigned int) width`" *casts* variable `width` to data type `unsigned int`). Unanticipated side effects of conversions are one source of integer-based vulnerabilities. C's rule for converting an integer to a wider type depends on the originating type. An unsigned integer is *zero-*

extended (0s fill the high-order bytes); a signed integer is *sign-extended* (the sign bit is propagated; this preserves signed values). Conversion to a smaller width truncates high-order bits. Same-width data type conversions between signed and unsigned integers do not alter any bits, but change interpreted values.

‡**Exercise** (C integer type conversion). Build an 8-by-8 table with rows and columns: s-char, u-char, s-short, u-short, s-int, u-int, s-long, u-long (s- is signed; u- unsigned). As entries, indicate conversion effects when sources (row headings) are converted to destinations (columns). Mark diagonal entries "same type". For each other entry include each of: value (changed, preserved); bit-pattern (changed, preserved); width-impact (sign-extended, zero-extended, same width, truncated). (Hint: [25, Chapter 6, page 229].)

INTEGER OVERFLOW IN C. Suppose x is an 8-bit `unsigned char`. It has range 0..255 (Table 6.1). If it has value `0xFF` (255) and is incremented, what value is stored? We might expect 256 (`0x100`), but that requires 9 bits. C retains the least significant 8 bits; x is said to *wrap around* to 0. This is an instance of *integer overflow*, which unsurprisingly, leads to programming errors—some exploitable. The issue is "obvious": exceeding the range of values representable by a fixed-width data type (and not checking for or preventing this). Since bounds tests on integer variables often dictate program branching and looping, this affects control flow. If the value of the variable depends on program input, a carefully crafted input may alter control flow in vulnerable programs.

Example *(Two's complement).* It helps to recall conventions for machine representation of integers. For an *unsigned integer*, a binary string $b_{n-1}b_{n-2}...b_1b_0$ is interpreted as a non-negative number in normal binary radix, with value $v = \sum_{i=0}^{n-1} b_i \cdot 2^i$. For *signed integers*, high-order bit b_{n-1} is a *sign bit* (1 signals negative). *Two's complement* is almost universally used for signed integers; for example in 4-bit two's complement (Table 6.3, page 164), incrementing binary 0111 to 1000 causes the value to wrap from $+7$ to -8.

Example *(Integer overflow: rate-limiting login).* Consider the pseudo-code:

```
handle_login(userid, passwd)          % returns TRUE or FALSE
  attempts := attempts + 1;           % increment failure count
  if (attempts <= MAX_ALLOWED)        % skip if over limit of 6
  { if pswd_is_ok(userid, passwd)     % if password is correct
    { attempts := 0;  return(TRUE); } %  reset count, allow login
  }                                   % else reject login attempt
  return(FALSE);
```

It aims to address online password guessing by rate limiting. Constant `MAX_ALLOWED` (6) is intended as an upper bound on consecutive failed login attempts, counted by global variable `attempts`. For illustration, suppose `attempts` were implemented as a 4-bit signed integer (two's complement). After six incorrect attempts, on the next one the counter increments to 7, the bound test fails, and `handle_login` returns FALSE. However if a persistent guesser continues further, on the next invocation after that, `attempts` increments from 7 (binary 0111) to 8 (binary 1000), which as two's complement is -8 (Table 6.3). The condition `(attempts <= MAX_ALLOWED)` is now TRUE, so rate limiting fails. Note that a test to check whether a seventh guess is stopped would falsely indicate that the program achieved its goal. (While C itself promotes to `int`, 16 bits or more, the problem is clear.)

MODULAR WRAPPING VS. UNDEFINED. To be more precise: for *unsigned* integer operands, the C standard officially declares that overflow does not exist: results that mathematically overflow an n-bit type are reduced modulo 2^n (truncated at n bits). In contrast for *signed* integers, C dictates that overflow (and underflow) results in undefined behavior—the operation is not illegal or prevented, but the result is machine-dependent. In practice, overflow of a signed integer typically wraps around to a negative value, and decrementing the largest representable negative integer results in wrap-around to the largest positive value. Other languages handle this differently, e.g., Python does automatic type promotion to larger-width data types.

INTEGER UNDERFLOW IN C. As qualified above, decrementing a negative integer below its smallest representable value wraps around to a positive in two's complement arithmetic; this is an *integer underflow*. Logic errors thus result from the (false) expectation that decreasing a negative number will never change its sign (and analogously for increasing a positive integer). The core C language has no built-in mechanism to prevent, check, raise alerts, or terminate programs on integer overflow or underflow.

Example *(Integer overflow on multiplication).* On a machine where an `int` is 16 bits, suppose that `width` and `height` are `unsigned int` C variables derived from user input. A program then dynamically allocates memory for an array of `width*height` elements. Attacker input results in these values being 235, 279. Now $235 * 279 = 65565 = 2^{16} + 29$ or `0x1001D`. To get a 16-bit result, C truncates to `001D`, and a malloc() intended to return memory for 65565 elements instead gets room for 29. The memory pointer returned, `bigtable`, is used to index elements `bigtable[i][j]`. This ends up referencing memory outside of that allocated for this data structure; C does not check this.

CATEGORIES OF INTEGER BUGS. Table 6.2 gives definitions and examples of five main categories of integer-based vulnerabilities: integer overflow, underflow, signedness mismatch, loss of information on narrowing, and value-change due to sign extension. These arise from arithmetic operations (e.g., $+, -, *, /$, <<, >>) on integer types,[2] plus moves or type conversions that widen (with zero-extension, sign-extension) or narrow (truncation) or change the binary interpretation between unsigned and two's complement.

C POINTER ARITHMETIC. C indexes arrays using a *subscript operator*: `b[i]` evaluates `((b)+(i))` in *pointer arithmetic*, then dereferences the resulting address to extract a value denoted `*(b+i)` as "pointer and offset". The integer expression `(i)` is computed with arithmetic conversion and promotion rules as above (negative results allowed). Pointer arithmetic means the offset is scaled: if `b` is defined to have 4-byte elements, `i` is multiplied by 4 before adding. C does not support adding two pointers, but two same-type pointers can be subtracted; the difference represents a number of elements (not an address difference). Unsurprisingly, security issues arise from the combination of this flexibility in pointer dereferencing, C allowing casting of an integer to a pointer, and unexpected integer values from arithmetic operations and conversions per Table 6.2.

SOFTWARE CONSEQUENCES. Integer-based vulnerabilities are indirect: issues arise

[2] "<<" is left-shift, e.g., multiply by 2; this may spill into the sign bit. Right-shift of a signed integer may be *logical* (0-filling the vacated sign bit) or *arithmetic* (sign-filling it); C leaves this machine-dependent.

Category	Description	Examples
integer overflow	value exceeds maximum representable in data type, e.g., >INT_MAX (signed) or >UINT_MAX	adding 1 to 16-bit UINT 0xFFFF yields not 0x10000, only low-order 16 bits 0x0000
		multiplying 16-bit UINTs produces a 16-bit result in C, losing high-order 16 bits
integer underflow	value below minimum representable in data type, e.g., (unsigned) <0 or (signed) <INT_MIN	subtracting 1 from signed char 0x80 (−128) yields 0x7F (+127), i.e., wraps
		subtracting 1 from 16-bit UINT 0x0000 (0) yields 0xFFFF (+65535)
signedness mismatch (same-width integers)	signed value stored into unsigned (or vice versa)	assigning 16-bit SINT 0xFFFE (−2) to 16-bit UINT will misinterpret value (+65534)
		assigning 8-bit UINT 0x80 (128) to 8-bit SINT changes interpreted value (−128)
narrowing loss	on assigning to narrower data type, truncation loses meaningful bits or causes sign corruption	assigning 32-bit SINT 0x0001ABCD to 16-bit SINT loses non-zero top half 0x0001
		assigning SINT 0x00008000 to 16-bit SINT gives representation error 0x8000 (-2^{15})
extension value change	sign extension of signed integer to wider unsigned (Beware: short, and also often char, are signed)	assigning signed char 0x80 to 32-bit UINT changes value to 0xFFFFFF80
		assigning 16-bit SINT 0x8000 to 32-bit UINT changes value to 0xFFFF8000

Table 6.2: Integer-based vulnerability categories. UINT, SINT are shorthand for unsigned, signed integer. Assignment-like conversions occur on integer promotion, casts, function parameters and results, and arithmetic operations. Table 6.3 reviews two's complement.

in later use of the integers. Failed sanity checks and logic errors result from variables having unexpected values. Exploitable vulnerabilities typically involve integer values that can be influenced by attacker input; many involve malloc(). Common examples follow.

1) Normal indexes (*subscripts*) within an array of n elements range from 0 to $n-1$. Unexpected subscript values resulting from integer arithmetic or conversions enable read and write access to unintended addresses. These are *memory safety violations*.
2) Smaller than anticipated integer values used as the size in memory allocation requests result in under-allocation of memory. This may enable similar memory safety violations, including buffer overflow exploits (Section 6.4).
3) An integer underflow (or other crafted input) that results in a negative size-argument to malloc() will be converted to an (often very large) unsigned integer. This may allocate an enormous memory block, trigger out-of-memory conditions, or return a NULL pointer (the latter is returned if requested memory is unavailable).
4) A signed integer that overflows to a large negative value may, if compared to an upper bound as a loop exit condition, result in an excessive number of iterations.

INTEGER BUG MITIGATION. While ALU flags (page 165) signal when overflows occur, these flags are not accessible to programmers from the C-language environment. If programming in assembly language, instructions could be manually inserted immediately after each arithmetic operation to test the flags and react appropriately (e.g., calling

Bitstring	Unsigned	one's complement	two's complement	Notes
0000	0	represents same value as unsigned		leftmost bit 0 signals positive integer; remaining bits specify magnitude
0001	1			
0010	2			
0011	3			
0100	4			
0101	5			
0110	6			
0111	7			
1000	8	−7	−8	leftmost bit is sign bit (1 if negative); one's complement has a redundant −0; two's complement has an extra value
1001	9	−6	−7	
1010	10	−5	−6	
1011	11	−4	−5	
1100	12	−3	−4	
1101	13	−2	−3	
1110	14	−1	−2	
1111	15	−0	−1	

Table 6.3: Interpretations of 4-bit strings as unsigned and signed integers. The magnitude of a negative bitstring is: for one's complement, the bitwise complement of the lower (rightmost) 3 bits; for two's complement, that value plus one.

warning or exit code). While non-standard, some C/C++ compilers (such as GCC, Clang) offer compile options to generate such instructions for a subset of arithmetic operations. A small number of CPU architectures provide support for arithmetic overflows to generate software interrupts analogous to memory access violations and divide-by-zero. In many environments, it remains up to developers and their supporting toolsets to find integer bugs at compile time, or catch and mitigate them at run time. Development environments and developer test tools can help programmers detect and avoid integer bugs; other options are binary analysis tools, run-time support for instrumented safety checks, replac-

Hardware result signed & unsigned	Unsigned interpretation, see carry flag	reason	CF	two's compl. interpretation, see OF	reason	overflow flag
1001 + 0001 = 1010	9 + 1 = 10	—	0	0	—	-7 + 1 = -6
0010 + 0111 = 1001	2 + 7 = 9	—	0	1	sign	2 + 7 = -7
1111 + 0010 = 0001	15 + 2 = 1	carry	1	0	—	-1 + 2 = 1
1001 + 1001 = 0010	9 + 9 = 2	carry	1	1	sign	-7 + -7 = 2
1001 − 0001 = 1000	9 − 1 = 8	—	0	0	—	-7 − 1 = -8
1001 − 0010 = 0111	9 − 2 = 7	—	0	1	sign	-7 − 2 = 7
0001 − 0010 = 1111	1 − 2 = 15	borrow	1	0	—	1 − 2 = -1
0010 − 1001 = 1001	2 − 9 = 9	borrow	1	1	sign	2 − -7 = -7

Table 6.4: Arithmetic operations and hardware flags CF (carry), OF (overflow), 4-bit examples. Despite being unaware of semantic intent, the same hardware addition instruction can be used whether the operands are unsigned or two's complement; similarly for subtraction. The flags—CF for unsigned operations, OF for signed operations—signal errors in computed results due to size-limited precision. The results meet hardware arithmetic specifications; software must check the flags to address any error cases.

ing arithmetic machine operations by calls to *safe integer library* functions, automated upgrading to larger data widths when needed, and arbitrary-precision arithmetic. None of the choices are easy or suitable for all environments; specific mitigation approaches continue to be proposed (Section 6.9). A complication for mitigation tools is that some integer overflows are *intentional*, e.g., programmers rely on wrap-around for functional results such as integer reduction modulo 2^{32}.

COMMENTS: INTEGER BUGS, POINTERS. We offer a few comments for context.

i) While we can debate C's choice to favor efficiency and direct access over security, our challenge is to deal with the consequences of C's installed base and wide use.

ii) Integer bugs relate to principle P15 (DATATYPE-VALIDATION) and the importance of validating all program input—in this case arithmetic values—for conformance to implicit assumptions about their data type and allowed range of values.

iii) How C combines pointers with integer arithmetic, and uses pointers (array bases) with the subscripting operation to access memory within and outside of defined data structures, raises memory safety and language design issues beyond P15 (and beyond our scope). Consequences include buffer overflow exploits (a large part of this chapter).

‡**Exercise** (Two's complement representation). Note the following ranges for n-bit strings: unsigned integer $[0, 2^n - 1]$; one's complement $[-(2^{n-1} - 1), 2^{n-1} - 1]$; two's complement $[-2^{n-1}, 2^{n-1} - 1]$. The n-bit string $s = b_{n-1}b_{n-2}...b_1b_0$ interpreted as two's complement has value: $v = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$. a) Verify that this matches the two's complement values in Table 6.3. b) Draw a circle as a clock face, but use integers 0 to $N-1$ to label its hours (0 at 12 o'clock); this shows how integers mod N wrap from $N-1$ to 0. c) Draw a similar circle, but now use labels 0000 to 1111 on its exterior and corresponding values $0, +1, ..., +7, -8, -7, ..., -1$ on its interior. This explains how overflow and underflow occur with 4-bit numbers in two's complement; compare to Table 6.3. d) To add 3 to 4 using this circle, step 3 units around the clock starting from 4. To add $-3 =$ `1101` (13 if unsigned) to 4, step 13 steps around the clock starting from 4 (yes, this works [30, §7.4]). This partially explains why the same logic can be used to add two unsigned, or two two's complement integers. e) The negative of a two's complement number is formed by bitwise complementing its string then adding 1; subtraction may thus proceed by negating the subtrahend and adding as in part d). Verify that this works by computing (in two's complement binary representation): $3 - 4$. (Negation of an n-bit two's complement number x can also be done by noting: $-x = 2^n - x$.)

‡**CARRY BIT, OVERFLOW BIT.** Some overflows are avoidable by software checks prior to arithmetic operations; others are better handled by appropriate action after an overflow occurs. Overflow is signaled at the machine level by two hardware flags (bits) that Arithmetic Logic Units (ALUs) use on integer operations: the *carry flag* (CF) and *overflow flag* (OF). (Aside: the word "overflow" in "overflow flag" names the flag, but the flag's semantics, below, differ from the typical association of this word with the events that set CF.) Informally, CF and OF signal that a result may be "wrong", e.g., does not fit in the default target size. Table 6.4 gives examples of setting these flags on addition and subtraction. (The flags are also used on other ALU operations, e.g., multiplication,

shifting, truncation, moves with sign extension; a third flag SF, the *sign flag*, is set if the most-significant bit of a designated result is 1.) CF is meaningful for unsigned operations; OF is for signed (two's complement). CF is set on addition if there is a carry out of the leftmost (most significant) bit, and on subtraction if there is a borrow into the leftmost bit. OF is set on addition if the sign bit reverses on summing two numbers of the same sign, and on subtraction if a negative number minus a positive gives a positive, or a positive minus a negative gives a negative. The same hardware circuit can be used for signed and unsigned arithmetic (exercise above); the flags signal (but do not correct) error conditions that may require attention. For example, in Table 6.4's third line, the flags differ: CF=1 indicates an alert for the unsigned operation, while OF=0 indicates normal for two's complement.

‡**Example** *(GCC option).* The compile option `-ftrapv` in the GCC C compiler is designed to instrument generated object code to test for overflows immediately after signed integer add, subtract and multiply operations. Tests may, e.g., branch to handler routines that warn or abort. Such insertions can be single instruction; e.g., on IA-32 architectures, `jo`, `jc` and `js` instructions jump to a target address if the most recent operation resulted in, respectively, an overflow (OF), carry (CF), or most-significant bit of 1 (SF).

6.3 Stack-based buffer overflows

Pouring in more water than a glass can hold causes a spill. If more bytes are written to a buffer or array than allocated for it, an analogous spill may overwrite content in adjacent memory. Such *buffer overflows* are not prevented in languages like C, and perhaps surprisingly, remain an ongoing issue on many platforms. A buffer overflow that occurs "naturally" (not intentionally) causes unpredictable outcomes—ranging from a system crash, or incorrect program output (sometimes unnoticed), to no ill effects at all (e.g., the memory overwritten is unused or irrelevant to program execution). Of greater interest is when an overflow is triggered with malicious intent, i.e., a *buffer overflow attack*.

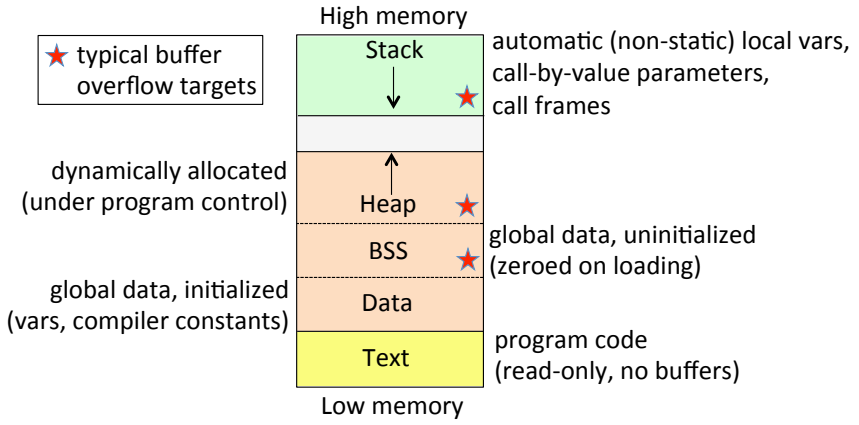
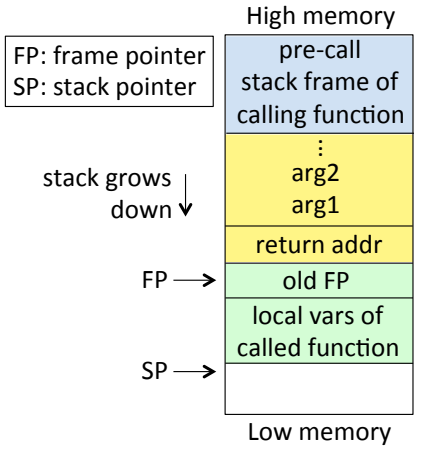


Figure 6.3: Common memory layout (user-space processes).

MEMORY LAYOUT (REVIEW). We use the common memory layout of Fig. 6.3 to explain basic concepts of memory management exploits. In Unix systems, environment variables and command line arguments are often allocated above "Stack" in this figure, with shared libraries allocated below the "Text" segment. *BSS* (*block started by symbol*) is also called the *block storage segment*. The *data segment* (BSS + Data in the figure) contains statically allocated variables, strings and arrays; it may grow upward with reorganization by calls to memory management functions.

STACK USE ON FUNCTION CALLS. *Stack-based buffer overflow* attacks involve variables whose memory is allocated from the stack. A typical schema for building stack frames for individual function calls is given in Fig. 6.4, with local variables allocated on the stack (other than variables assigned to hardware registers). Reviewing notes from a background course in operating systems may help augment this summary overview.



1. calling function pushes args onto stack
2. "call" opcode pushes Instruction Pointer (IP) as return address, then sets IP to begin executing code in called function
3. called function pushes FP for later recovery
4. FP ← SP (so FP points to old FP), now FP+k = args, FP-k = local vars
5. decrement SP, making stack space for local vars
6. called function executes until ready to return
7. called function cleans up stack before return (SP ← FP , FP ← old FP popped from stack)
8. "ret" opcode pops return address into IP, to resume execution back to calling function

Figure 6.4: User-space stack and function call sequence (x86 conventions). FP is also called BP (Base Pointer). Register state may also be saved onto the stack (not shown).

Example *(Buffer overflow).* With memory layout per Fig. 6.4, and a machine with 4-byte memory words, consider this contrived C function to illustrate concepts:

```
void myfunction(char *src)    /* src is a ptr to a char string */
{  int var1, var2;            /* 1 stack word used per integer */
   char var3[4];              /* also 1 word for 4-byte buffer */

   strcpy(var3, src);         /* template:  strcpy(dst, src)   */
}
```

Figure 6.5 shows the stack frame once this function is called. Local variables allocated on the stack include `var3`; it can hold a character string of length 3, plus a final NUL byte (`0x00`) to signal end-of-string, per C convention. The C library string-copy routine, `strcpy()`, copies byte for byte from source to destination address, stopping only after copying a string-terminating NUL. If n denotes the length of the string that `src` points to, then if $n > 3$ a buffer overflow occurs, and memory at addresses higher than (above) `var3` will be overwritten, marching through `var2` towards and past the frame's return ad-

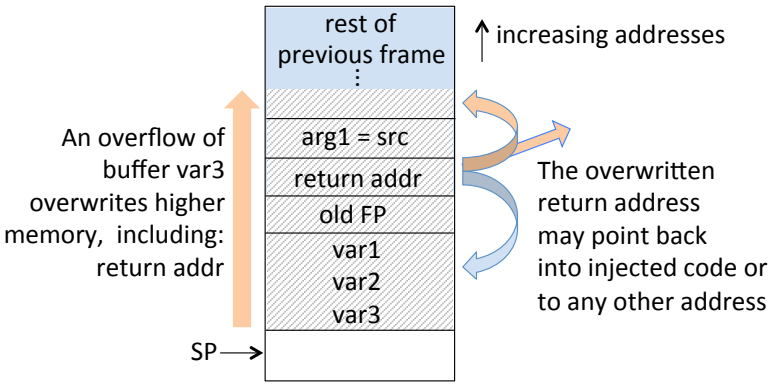


Figure 6.5: Buffer overflow of stack-based local variable.

dress if *n* is large enough. When `myfunction()` returns, the Instruction Pointer (Program Counter) is reset from the return address; if the return address value was overwritten by the string from `src`, program control still transfers to the (overwriting) value. Now suppose the string `src` came from malicious program input—both intentionally longer than `var3`, and with string content specifically created (by careful one-time effort) to overwrite the stack return address with a prepared value. In a common variation, this value is an address that points back into the stack memory overwritten by the overflow of the stack buffer itself. The Instruction Pointer then retrieves instructions for execution from the (injected content of the) stack itself. In this case, if the malicious input (a character string) has binary interpretation that corresponds to meaningful machine instructions (opcodes), the machine begins executing instructions specified by the malicious input.

NO-OP SLED. Among several challenges in crafting injected code for stack execution, one is: 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 overwrite 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*.[3] 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 NO-OP sled is a telltale sign of an attack, attackers may replace literal NOP instructions with equivalent instructions having no effect (e.g., OR 0 to a register). This complicates sled discovery.

6.4 Heap-based buffer overflows and heap spraying

Beyond the stack, overflows may affect buffers in heap memory and the data segment (BSS and Data in Fig. 6.3). Traditionally, many systems have left the heap and BSS not only writable (necessary), but also executable (unnecessary, dangerous). The data

[3]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).

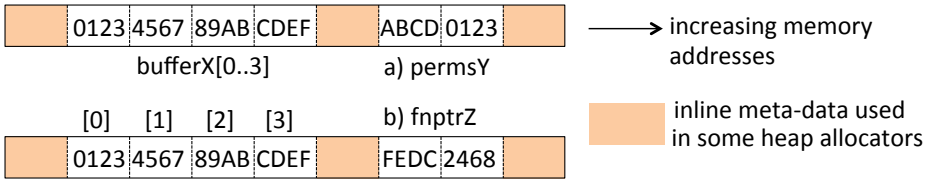


Figure 6.6: Heap-based buffer overflow. Writing past the end of a heap-allocated buffer can overwrite adjacent heap-allocated variables. a) A permissions-related variable may be overwritten. b) A function pointer may be overwritten. Both cases highlight that program decisions are affected not only by program code itself, but also by data.

segment can also be subdivided into read-only (e.g., for constants) and read-write pieces. While here we focus on heap-based exploits, data in any writable segment is subject to manipulation, including environment variables and command-line arguments (Fig. 6.3). A buffer is allocated in BSS using a C declaration such as: `static int bufferX[4]`.

OVERFLOWING HIGHER-ADDRESS VARIABLES. How dynamic memory allocation is implemented varies across systems (stack allocation is more predictable); attackers experiment to gain a roadmap for exploitation. Once an attacker finds an exploitable buffer, and a strategically useful variable at a nearby higher memory address, the latter variable can be corrupted. This translates into a tangible attack (beyond denial of service) only if corruption of memory values between the two variables—a typical side effect—does not "crash" the executing program (e.g., terminate it due to errors). Figure 6.6 gives two examples. In the first, the corrupted data is some form of access control (permission-related) data; e.g., a FALSE flag might be overwritten by TRUE. The second case may enable overwriting a *function pointer* (Fig. 6.7) holding the address of a function to be called. Overwriting function pointers is a simple way to arrange control transfer to attacker-selected code, whereas simple stack-based attacks use return addresses for control transfer.

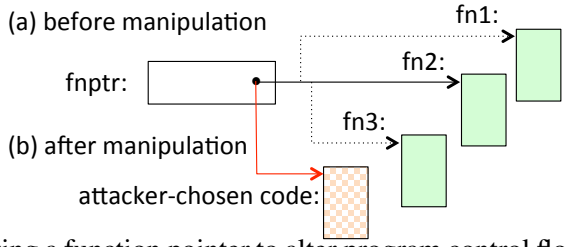


Figure 6.7: Corrupting a function pointer to alter program control flow. How the attacker-chosen code is selected, or injected into the system, is a separate issue.

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:

a) stack-based pointers, including return addresses and frame pointers;

b) function pointers (allocated in the stack, heap or static area), including in any function address lookup table (*jump table*, dispatch table, virtual table or *vtable*); and

c) addresses used in C-language `setjmp`/`longjmp` functions. (These are used in non-standard call sequences, such as for exception-handling or co-routines.) Also:

d) (indirectly) by corrupting data used in a branching test; however, the branch is to a fixed address. Section 6.2 contains a related example involving integer overflows.

GENERIC EXPLOIT STEPS. Having considered stack- and heap-based attacks, note that many buffer overflow and related exploits involve three functional steps:

1. *Code injection or location*. Code that the attacker desires to be executed is somehow placed within the target program's address space. If existing system utilities or other code meet the attacker's goal, injection is not needed, just the address of the code.
2. *Corruption of control flow data*. One or more data structures is overwritten, e.g., by a buffer overflow corrupting adjacent data. This may be separate from or part of step 1. The corruption sets up later transfer of control, directly to the step 1 address if known.
3. *Seizure of control*. Program control flow is transferred to the target code of step 1. This may be by simply waiting after having engineered the transfer by step 2.

HEAP SPRAYING. *Heap spraying* is a method that places into the heap a large number of instances of attacker-chosen code (Figure 6.8). This achieves step 1 above; an independent exploit is relied on for step 2. It has been a popular means to exploit browsers in *drive-by download* attacks (Chapter 7). The attack allocates a large number (e.g., thousands) of heap objects, their content chosen by the attacker. This might consume 1–100 megabytes. It is arranged, e.g., by embedding into HTML pages served by a web site, a script that allocates 10,000 strings in a simple JavaScript loop, assigning to each element of an array a constant string whose bytes are an opcode sequence of a long no-op sled (Section 6.3) followed by *shellcode* (Section 6.8). An alternative is to arrange that rendering a visited web page loads an image that results in the end-user machine allocating similar heap objects. Step 2 (above) may use any means (e.g., corrupting a function pointer) that transfers program control to the heap. The more objects and the longer the no-op sleds, the higher the probability that a transfer to an arbitrary heap address will hit a sled to slide into the shellcode. In selecting the address pointing into the heap, attackers use knowledge of typical memory layout to increase their success probability. Note that this attack need not involve a buffer overflow, often uses JavaScript (a type-safe language), and is not stopped by defenses that randomize heap layout (Section 6.6).

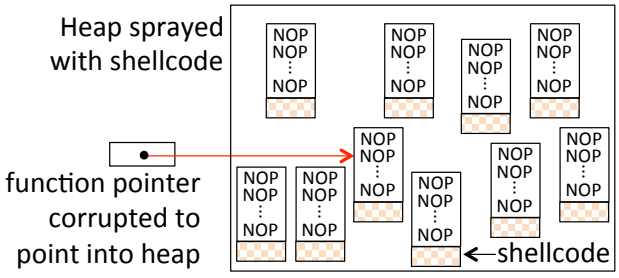


Figure 6.8: Heap spraying. Once the program utilizes the corrupted function pointer, the Instruction Pointer begins retrieving code for execution from the heap.

‡**Exercise** (Exploiting heap maintenance). C-family programs allocate dynamic memory using malloc(), with underlying system calls that manage heap memory in blocks or *chunks*. For heaps with inline metadata, each chunk starts with a header field indicating whether the chunk is free or allocated, and a next-chunk pointer using a singly or doubly linked list. Overflowing a buffer allocated from such heap memory can overwrite pointers as explained above. a) Summarize how this enables malicious overwriting of arbitrary memory locations (hint: [5]). b) Discuss *secure heap allocator* defenses (hint: Sect. 6.9).

‡**Exercise** (Format string vulnerabilities). A class of attacks distinct from overflows and heap spraying exploits how *format strings* interact with system memory management in function families such as C's `printf(format, string)`. When the first argument format string is a dynamic variable (rather than a constant like `"%s"`), the rich functionality provided enables reading and writing at arbitrary memory addresses. Look up and explain format string attacks and defenses (hint: [49] or [7, pages 125-128], and Section 6.9).

6.5 ‡Return-to-libc exploits

Some buffer overflow attacks inject code into the run-time stack or heap memory, and then execute that code. As noted in Section 6.6, such attacks can be stopped if support for *non-executable memory* ranges is available and utilized. However, this defense does not stop *return-to-libc* attacks, described next.

STACK-BASED RETURN-TO-LIBC ATTACK. Such an attack may proceed as follows. A return address is overwritten as in stack-based attacks above, but now it is pointed to transfer execution not to new 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, which then executes the command using execl() per Section 6.8, 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 6.9.

RETURN-TO-LIBC WITH STRCPY OF SHELLCODE. In a second example of a return-to-libc attack that defeats non-executable stacks, where system() was used above, now strcpy() is called. It copies a string from a specified source to a specified destination address; these addresses are part of the data injected into the buffer, now arranged as stack arguments for strcpy(). The injected data also includes *shellcode* (Section 6.8), and the call to strcpy() copies that from the (non-executable) stack to a location in a segment that is both writable and executable (perhaps heap; the Text segment is now commonly non-writable). The stack return address that will be used by strcpy(), in place of the address of

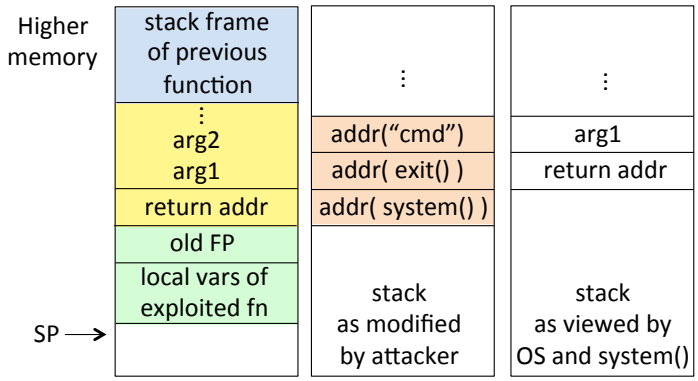


Figure 6.9: Return-to-libc attack on user-space stack. A local stack variable is overflowed such that the return address becomes that of the library call system(), which will expect a "normal" stack frame upon entry, as per the rightmost frame. If what will be used as the return address upon completion 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.

exit() in Figure 6.9, is set to also point to the destination address for the shellcode. This results in the shellcode being executed on the return from strcpy().

6.6 Buffer overflow exploit defenses and adoption barriers

BUFFER OVERFLOW COUNTERMEASURES. Various measures can counter buffer overflow attacks. As a general distinction, compile-time techniques reduce the number of vulnerabilities by changing software before deployment (e.g., compiler tools flag potential issues for developers to examine), while others involve run-time mechanisms to prevent exploitation of vulnerabilities; some defenses combine these. The former often impose extra work on developers, while the latter incur run-time overhead and changes to run-time support. Well-known approaches include the following, among a number of others.

1) **NON-EXECUTABLE STACK AND HEAP.** Buffer overflow attacks that execute injected code directly on the stack or heap itself can be stopped if support exists to flag specified address ranges as *non-executable memory*. Address ranges assigned to the stack, heap and BSS can then be marked invalid for loading via the Instruction Pointer (Program Counter). More generally, *data execution prevention* (DEP) techniques may be provided by hardware (with an *NX bit*) or software. However, support being available does not guarantee its use—e.g., due to accommodating backwards compatibility, being disabled by an attacker, or use of just-in-time (JIT) run-time systems that require an executable heap. Also, DEP prevents execution but not overwriting of memory itself, and thus does not stop all attacks involving *memory safety violations*.
2) **STACK PROTECTION (RUN-TIME).** *Stack canaries* are checkwords used to detect code injection. An extra field is inserted in stack frames just below (at lower address than) attack targets such as return addresses—in Fig. 6.4, just above the local variables. A buffer overflow attack that corrupts all memory between the buffer and the return

address will overwrite the canary. A run-time system check looks for an expected (canary) value in this field before using the return address. If the canary word is incorrect, 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 XORing a secret mask, so that attacks that overwrite the pointer corrupt it but cannot usefully modify the control flow).

3) **RUN-TIME BOUNDS-CHECKING.** Here, compilers instrument code to invoke run-time support that tracks, and checks for conformance with, bounds on buffers. This involves compiler support, run-time support, and run-time overhead.

4) **MEMORY LAYOUT RANDOMIZATION (RUN-TIME).** Code injection attacks require precise memory address calculations, and often rely on predictable (known) memory layouts on target platforms. To disrupt this, defensive approaches (including ASLR) randomize the layout of objects in memory, including the base addresses used for run-time stacks, heaps, and executables including run-time libraries. Some *secure heap allocators* include such randomization aspects and also protect heap metadata.

5) **TYPE-SAFE LANGUAGES.** Operating systems and system software (distinct from application software) have historically been written in C. Such systems languages allow *type casting* (converting between data types) and unchecked *pointer arithmetic* (Section 6.2). These features contribute to 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. Programming languages with weak data-typing violate principle P15 (DATATYPE-VALIDATION).

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; and by efficiency-driven convention, the presence of a NUL 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. Thus various proposals promote use of *safe C libraries* to replace the historical libc, whose string-handling functions lack bounds-checks. One approach is to instrument compiler warnings instructing programmers to use substitute functions; this of course does not patch legacy code.

7) **STATIC ANALYSIS TOOLS (COMPILE-TIME, BINARIES).** If the vulnerable code itself did bounds-checking, many buffer overflow errors would be avoided. Thus an available defense is to train developers to do bounds-checking, and support this by encouraging use of compile-time tools, e.g., *static analysis tools* that flag memory management vulnerabilities in source code for further attention. Binaries can also be analyzed. (Aside: even if adopted, such tools miss some vulnerabilities, and raise false alarms. Discussion of *dynamic analysis* and related approaches is beyond our scope.)

‡**Exercise** (Control flow integrity). Summarize how compile-time (static) analysis can

be combined with run-time instrumentation for *control flow integrity*, stopping program control transfers inconsistent with compile-time analysis (hint: [1, 2]; cf. [12]).

ADOPTION BARRIERS. The above list of countermeasures to buffer overflow attacks, while incomplete, suffices to highlight both a wide variety of possible approaches, 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 the following.

i) *No single governing body.* While some standards groups are influential, no corporation, country, government, or organization has the power to impose and enforce rules on all software-based systems worldwide, even if ideal solutions were known.

ii) *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, face immediate opposition.

iii) *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 us 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 (page 160) C, C++ and assembler, makes the idea of modifying or replacing all such software impractical, for multiple reasons: cost, lack of available expertise, unacceptability of disrupting critical systems. Nonetheless, much progress has been made, with various approaches now available to mitigate exploitation of memory management vulnerabilities, e.g., related to buffer overflows.

‡**Exercise** (Case study: buffer overflow defenses). Summarize the effectiveness of selected buffer overflow defenses over the period 1995-2009 (hint: [56]).

6.7 Privilege escalation and the bigger picture

A typical attack may proceed as follows. On a victim machine, an attacker first gets some code of her choosing (or under her control) to run, e.g., by exploiting a buffer overflow—call it *base-camp* code. If this cannot itself accomplish the attack end-goal, it is used as a *stepping stone* to run other programs, transfer control to other routines, or spawn new processes. If the base-camp code has insufficient flexibility or privileges to achieve the end-goal, *privilege escalation* is also used—changing the execution environment (or protection domain, Chapter 5) in some way to reduce constraints or increase privileges, as described below. Privilege escalation is related to principle P6 (LEAST-PRIVILEGE); a process with fewer privileges results in less damage when compromised.

FORMS OF ESCALATION. Example levels of ability or privilege escalation are:

i) moving from the fixed functionality of a compiled program to a shell allowing execution of arbitrary commands and other programs;

ii) moving from an isolated "sandbox" to having access to a complete filesystem;

iii) moving from a non-root process to code running with UID 0; and

iv) moving from UID 0 privileges (user-space process) to kernel-mode privileges.

For item ii), the isolated environment might be, e.g., a filesystem jail (Chapter 5) or a browser sandbox (preventing the browser from accessing local files other than in */tmp*). An example of iii) involves exploiting file access race conditions (Section 6.1).

Example *(Escalation via root-owned setuid).* One path to a root shell is to find a root-owned setuid program vulnerable to buffer overflow injection. Consider these steps:

1. An attacker has local access via a user-space process (e.g., shell) on a target machine. The machine hosts a vulnerable root-owned setuid program as noted.
2. A crafted input including shellcode (Section 6.8) is supplied to the setuid program, e.g., as command line input from this shell. The input overflows a buffer in such a way that injected code executes and spawns a new shell. It will run as root.[4]
3. The attacker may now type commands into the root shell (via `stdin`), redirect scripted commands from a stored file, or retrieve commands over the network.

PRIVILEGES AND TCP/IP PORTS. Some systems declare TCP and UDP ports 0-1023 to be *privileged ports*—a process must run as root to *bind* to a privileged port and provide services (open it to listen for and process packets sent to it).[5] In turn, processes running on privileged ports are trusted—parties connecting to them expect non-malicious services, and safety in belief that the host machine owner allows only trustworthy processes to serve these ports. If a root-privileged *network daemon* (background process receiving network packets) is vulnerable to a base-camp attack, superuser privileges are easily gained—e.g., it suffices to execute, on the daemon code, a buffer overflow injection that spawns a new shell. That shell inherits the daemon's UID and associated privileges (Section 6.8).

COMMON FAILURES TO LIMIT PRIVILEGES. The above-noted trust in services on privileged ports is at times misplaced. For example, on any personal computer, regular work should be done using non-root accounts, with root used only when necessary (e.g., for configuration or software installation). This provides protection against a user unintentionally deleting, for example, the entire filesystem. However, often for convenience or due to oversight, a root account is used for regular operations (perhaps by continued use of an original default account); then compromise of any process run under that account surrenders superuser privileges. Similarly, despite best practices dictating that only well-scrutinized, time-tested programs be bound to privileged ports, vulnerable programs may be, increasing exposure to privilege escalation.

‡**Example** *(Remote-access shellcode).* Above, the buffer overflow attack provided a root shell to a local-access attacker. The procedure is similar for a network daemon (e.g., SSH or FTP service) exploited by a remote attacker to spawn a root shell, but now the victim machine communicates data to the remote attacker over the TCP/IP connection as usual for remote users. By convention, as a child process, the new shell inherits the standard streams (`stdin`, `stdout`) of its parent, and network connectivity is maintained.

[4] Chapter 5 explains how process UID affects privileges. Section 6.8 reviews process creation details.

[5] *Linux capabilities* [29] are privilege units; CAP_NET_BIND_SERVICE allows binding to ports 0–1023.

6.8 ‡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. Note that Section 5.4 (setuid programs) also discusses fork(), exec(), and inheriting parent UIDs in process creation.

SHELLCODE (TERMINOLOGY). A *command shell* provides a well-defined interface to system utility services. From it, arbitrary system commands or other programs can be run by specifying the program name (filepath if needed) and arguments. A common attack goal is to obtain a shell running with UID of root (*root shell*), 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. Used narrowly, the term *shellcode* refers to a short sequence of injected code that creates a command shell when executed—ideally a root shell; 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). System calls thus run in supervisor mode, and are how kernel resources are accessed. (Recall: *supervisor* refers to a privileged CPU hardware mode, allowing access to restricted machine instructions; in contrast, a *superuser* (root) process runs in user space like other user processes, albeit with more privileges, e.g., to run executables that regular processes cannot.)

COMMAND SHELLS AND FORK (BACKGROUND). An OS *command line interpreter* (shell) is not part of the core OS, but provides access to many OS features and is a main

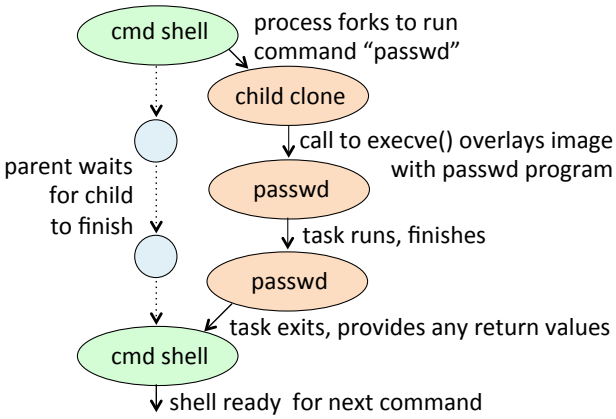


Figure 6.10: Command shell forking a child process to execute a command.

system interface for users; graphical user interfaces (GUIs) are an alternative. 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 command 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 (Fig. 6.10). This proceeds on Unix by calling fork(), which clones the calling process; the clone recognized as the child (Chapter 5) then calls execve() to replace its image by the desired program to run the user-requested command. 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 SHELL (BACKGROUND). Sample C code to create an interactive shell is:

```
char *name[2];
   name[0] = "sh";      /* NUL denotes a byte with value 0x00 */
   name[1] = NULL;      /* NULL denotes a pointer of value 0  */
   execve("/bin/sh", name, NULL);
```

We view execve() as the model library wrapper to the exec() syscall, with general form:

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

Here `path` (pointer to string) is the pathname of the file to be executed; `"v"` in the name `execve` 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 variables (each entry a pointer to a NUL-terminated string name=value); 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 PID/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 calls may essentially be mapped onto execve(). One is `execl(path, arg0, ...)` where `"l"` is mnemonic for *list*, and individual arguments beyond `path` are in a NULL-ended list of pointers to NUL-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` 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 challenges within injected code are: *eliminating NUL bytes* (`0x00`), and *relative addressing*. NUL 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 NUL byte in any opcode as end-of-string. This issue is overcome by use of alternative instructions and code sequences avoiding opcodes containing `0x00`. Relative addressing (within injected shellcode) is necessary for position-independent code, as the address at which shellcode will itself reside is not known—but standard coding practices address this, using (Program Counter) PC-relative addressing where supported; on other architectures (e.g., x86), a machine register is loaded with the address of an anchor shellcode instruction, and machine operations use addressing relative to the register. Overall, once one expert figures out shellcode details, automated tools allow easy replication by others.

6.9 ‡End notes and further reading

Dowd [25] is highly recommended for broad coverage of software security, including detailed examples of race conditions and integer bugs. Among early *white-hat* software security books (focused mainly on protection, to avoid security-related programming design and implementation errors), Howard [32] gives an extended treatment with focus on Windows environments; a later shorter offering [33] highlights common software security errors. In complementary books advancing software security as a subdiscipline, Viega [62] provides a white-hat compilation while the follow-up *black-hat* book (offensively-focused, with code-level attack details) with Hoglund [31] has extended discussion of *attack patterns*. Other black-hat treatments are Anley [4] (for shellcode) and McClure [41]. The Section 6.8 shellcode is from `mudge` [43], predating the stack-based buffer overflow roadmap of `Aleph One` [3]; these and the heap-based buffer overflow tutorial of Conover [18] are classic black-hat papers, along with a *Phrack* article [5] discussing techniques to exploit malloc()-managed memory. See Tanenbaum [58] for crisp background on operating systems, command shells and system calls.

See Bishop [9] for TOCTOU problems involving filesystem races; Tsafrir [59] offers a portable solution for a subset of file race conditions, but Cai [14] shows they can be defeated, and recommends known non-portable (but secure) defenses. Payer's solution [45] for file-based race conditions automatically replaces existing system calls with safe calls that cache metadata for accessed files. Chari [15] mitigates filename-based privilege escalation by focusing on safe *pathname resolution* mechanisms, rather than on race conditions per se; Vijayakumar [63] also provides means to find vulnerabilities in resolving names to resource references. For extended discussion of how to define and distinguish filepaths that are safe to resolve, see also Kupsch [38]. To address *integer-based vulnerabilities*, a proposal by Brumley [10] uses compiler-instrumented code providing run-time checks; similarly, Wurster [66] proposes efficient ARM-specific instruction sequences to respond to hardware flags signaling overflows and carries. See Regehr [24] for an empirical analysis of integer overflows in C/C++, and Kernighan [36] for C background. Hamacher [30, §7.4] explains why two's complement is preferred for representing signed numbers in logic circuits, and why the same circuit can serve two's complement and unsigned operands.

For early surveys on buffer overflow defenses, see Wilander [65] and Cowan [22]. For systematic studies of memory safety and memory corruption bugs, see van der Veen [61] and Szekeres [57]; similarly for *control flow integrity* specifically, see Burow [12]. Dereferencing *dangling pointers* (pointers to already freed memory) results in *use-after-free* errors, or *double-free errors* if freed a second time, both leading to memory safety violations; for defenses, see Caballero [13] and Lee [39], and *secure heap allocators* (Silvestro [53] provides references) to protect heap metadata, including by tactics similar to ASLR (below). For run-time bounds checking, see Jones [35]. Miller's improvements [42] to C string handling libraries have enjoyed adoption (but not by the GNU C library). For *memory-safe dialects* of C (these require code porting and run-time support), see CCured [44] and Cyclone [34]. For stack canaries, see StackGuard [21], and its extension PointGuard [20], which puts canaries next to code pointers, including in heap data. Forrest [28] proposed 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, but attacks remain [51]. Keromytis [37] surveys other proposals to counter code injection using randomization, including *instruction set randomization*. On *format string vulnerabilities*, see the black-hat exposition by `scut` [49]; for defenses, see Shankar [52] and FormatGuard [19]. Shacham's collection [50, 11, 47] explains return-to-libc attacks [55] and *return-oriented programming* (ROP) generalizations; see also Skowyra [54]. For *heap spraying* and defenses, see NOZZLE [46] and ZOZZLE [23].

For static analysis to detect buffer overruns, see Wagner [64]; see also Engler [26, 6], and a summary of Coverity's development of related tools [8]. A *model-checking* security analysis tool called MOPS (MOdel Checking Programs for Security) [17] encodes rules for safe programming (e.g., temporal properties involving ordered sequences of operations), builds a model, and then uses compile-time program analysis to detect possible rule violations. The WIT tool (Write Integrity Testing) [2] protects against memory error exploits, combining static analysis and run-time instrumentation. For discussion of vulnerability assessment, penetration testing and fuzzing, see Chapter 11. For *manual code inspection*, see Fagan [27]. For evidence that shellcode may be difficult to distinguish from non-executable content, see Mason [40].

References (Chapter 6)

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Comp. & Comm. Security (CCS)*, pages 340–353, 2005. 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 (Elias Levy). Smashing the stack for fun and profit. In *Phrack Magazine*. 8 Nov 1996, 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 (2nd edition)*. Wiley, 2007.

[5] anonymous. Once upon a free()... In *Phrack Magazine*. 11 Aug 2001, vol.11 no.57, file 9 of 18, `http://www.phrack.org` (for summaries see: Dowd [25, p. 184-186], Aycock [7, p. 119-123]).

[6] K. Ashcraft and D. R. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symp. Security and Privacy*, pages 143–159, 2002.

[7] J. Aycock. *Computer Viruses and Malware*. Springer Science+Business Media, 2006.

[8] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Comm. ACM*, 53(2):66–75, 2010.

[9] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, 1996.

[10] D. Brumley, D. X. Song, T. Chiueh, R. Johnson, and H. Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *Netw. Dist. Sys. Security (NDSS)*, 2007.

[11] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *ACM Comp. & Comm. Security (CCS)*, pages 27–38, 2008.

[12] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys*, 50(1):16:1–16:33, 2017.

[13] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Int'l Symp. Soft. Testing & Anal. (ISSTA)*, pages 133–143, 2012.

[14] X. Cai, Y. Gui, and R. Johnson. Exploiting Unix file-system races via algorithmic complexity attacks. In *IEEE Symp. Security and Privacy*, pages 27–44, 2009.

[15] S. Chari, S. Halevi, and W. Z. Venema. Where do you want to go today? Escalating privileges by pathname manipulation. In *Netw. Dist. Sys. Security (NDSS)*, 2010.

[16] H. Chen, D. Dean, and D. A. Wagner. Model checking one million lines of C code. In *Netw. Dist. Sys. Security (NDSS)*, 2004.

[17] H. Chen and D. A. Wagner. MOPS: An infrastructure for examining security properties of software. In *ACM Comp. & Comm. Security (CCS)*, pages 235–244, 2002. See also [16], [48].

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

[19] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX Security*, 2001.

[20] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security*, 2003.

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

[22] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Info. Survivability Conf. and Expo (DISCEX)*, Jan. 2000.

[23] C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In *USENIX Security*, 2011.

[24] W. Dietz, P. Li, J. Regehr, and V. S. Adve. Understanding integer overflow in C/C++. *ACM Trans. Softw. Eng. Methodol.*, 25(1):2:1–2:29, 2015. Shorter conference version: *ICSE* 2012.

[25] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2006.

[26] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Operating Sys. Design & Impl. (OSDI)*, pages 1–16, 2000.

[27] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[28] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *IEEE HotOS*, 1997.

[29] S. E. Hallyn and A. G. Morgan. Linux capabilities: making them work. In *Linux Symp.*, July 2008.

[30] V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky. *Computer Organization*. McGraw-Hill, 1978.

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

[32] M. Howard and D. LeBlanc. *Writing Secure Code (2nd edition)*. Microsoft Press, 2002.

[33] M. Howard, D. LeBlanc, and J. Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, 2009.

[34] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conf.*, pages 275–288, 2002.

[35] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1995. Original July 1995 announcement "Bounds Checking for C", `https://www.doc.ic.ac.uk/~phjk/BoundsChecking.html`.

[36] B. Kernighan and D. Ritchie. *The C Programming Language, 2/e*. Prentice-Hall, 1988. (1/e 1978).

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

[38] J. A. Kupsch and B. P. Miller. How to open a file and not get hacked. In *Availability, Reliability and Security (ARES)*, pages 1196–1203, 2008. Extended version: `https://research.cs.wisc.edu/mist/papers/safeopen.pdf`.

[39] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. In *Netw. Dist. Sys. Security (NDSS)*, 2015.

[40] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *ACM Comp. & Comm. Security (CCS)*, pages 524–533, 2009.

[41] S. McClure, J. Scambray, and G. Kurtz. *Hacking Exposed 6: Network Security Secrets and Solutions (6th edition)*. McGraw-Hill, 2009.

[42] T. C. Miller and T. de Raadt. strlcpy and strlcat - consistent, safe, string copy and concatenation. In *USENIX Annual Technical Conf.*, pages 175–178, 1999. FREENIX track.

[43] mudge (Peiter Zatko). How to write Buffer Overflows. 20 Oct 1995, available online.

[44] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[45] M. Payer and T. R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Virtual Execution Environments (VEE)*, pages 215–226, 2012.

[46] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *USENIX Security*, pages 169–186, 2009.

[47] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Systems and Security*, 15(1):2:1–2:34, 2012.

[48] B. Schwarz, H. Chen, D. A. Wagner, J. Lin, W. Tu, G. Morrison, and J. West. Model checking an entire Linux distribution for security violations. In *Annual Computer Security Applications Conf. (ACSAC)*, pages 13–22, 2005.

[49] scut / team teso. Exploiting Format String Vulnerabilities (version 1.2). 1 Sept 2001, online; follows a Dec. 2000 Chaos Communication Congress talk, `https://events.ccc.de/congress/`.

[50] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Comp. & Comm. Security (CCS)*, pages 552–561, 2007.

[51] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM Comp. & Comm. Security (CCS)*, pages 298–307, 2004.

[52] U. Shankar, K. Talwar, J. S. Foster, and D. A. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security*, 2001.

[53] S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu. Guarder: A tunable secure allocator. In *USENIX Security*, pages 117–133, 2018. See also "FreeGuard" (*CCS* 2017) for heap allocator background.

[54] R. Skowyra, K. Casteel, H. Okhravi, N. Zeldovich, and W. W. Streilein. Systematic analysis of defenses against return-oriented programming. In *Reseach in Attacks, Intrusions, Defenses (RAID)*, 2013.

[55] Solar Designer. "return-to-libc" attack. Bugtraq, Aug. 1997.

[56] A. Sotirov. Bypassing memory protections: The future of exploitation. USENIX Security (talk), 2009. `https://www.usenix.org/legacy/events/sec09/tech/slides/sotirov.pdf`, video online.

[57] L. Szekeres, M. Payer, T. Wei, and R. Sekar. Eternal war in memory. *IEEE Security & Privacy*, 12(3):45–53, 2014. Longer systematization (fourth author D. Song) in *IEEE Symp. Sec. and Priv.* 2013.

[58] A. S. Tanenbaum. *Modern Operating Systems (3rd edition)*. Pearson Prentice Hall, 2008.

[59] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably solving file TOCTTOU races with hardness amplification. In *USENIX File and Storage Tech. (FAST)*, 2008. Also: *ACM Trans. on Storage*, 2008.

[60] E. Tsyrklevich and B. Yee. Dynamic detection and prevention of race conditions in file accesses. In *USENIX Security*, 2003.

[61] V. van der Veen, N. dutt-Sharma, L. Cavallaro, and H. Bos. Memory errors: The past, the present, and the future. In *Reseach in Attacks, Intrusions, Defenses (RAID)*, pages 86–106, 2012.

[62] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2001.

[63] H. Vijayakumar, J. Schiffman, and T. Jaeger. STING: Finding name resolution vulnerabilities in programs. In *USENIX Security*, pages 585–599, 2012. See also Vijayakumar, Ge, Payer, Jaeger, "JIGSAW: Protecting resource access by inferring programmer expectations", *USENIX Security* 2014.

[64] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Netw. Dist. Sys. Security (NDSS)*, 2000.

[65] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Netw. Dist. Sys. Security (NDSS)*, 2003.

[66] G. Wurster and J. Ward. Towards efficient dynamic integer overflow detection on ARM processors. Technical report, BlackBerry Limited, Apr. 2016.