Chapter 8

Code Security

Software security is hard

- One of the main problems in software engineering at the moment
  - often poor programming because of lacking education/awareness in developers and bad tooling (languages/platforms making mistakes too easy to make and impact of mistakes too severe)
  - often due to project deadlines
- Unclear how to practically write correct and secure code, even with increased project resources
  - formal validation is extremely costly, not clear how to do on complex code bases
- Therefore many security relevant errors in currently deployed code
- Classification of security problems: “Common Weakness Enumeration” (CWE) at https://cwe.mitre.org/
- Publicly known software vulnerabilities: “Common Vulnerabilities and Exposures” (CVE) at https://cve.mitre.org/

CWE/SANS Top 25 most dangerous software errors

Insecure Interaction Between Components

- CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
- CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
- CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CWE-434 Unrestricted Upload of File with Dangerous Type
- CWE-352 Cross-Site Request Forgery (CSRF)
- CWE-601 URL Redirection to Untrusted Site ('Open Redirect')

http://www.sans.org/top25-software-errors/

CWE/SANS Top 25 most dangerous software errors

Risky Resource Management

- CWE-120 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
- CWE-22 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
- CWE-494 Download of Code Without Integrity Check
- CWE-829 Inclusion of Functionality from Untrusted Control Sphere
- CWE-676 Use of Potentially Dangerous Function
- CWE-131 Incorrect Calculation of Buffer Size
- CWE-134 Uncontrolled Format String
- CWE-190 Integer Overflow or Wraparound

http://www.sans.org/top25-software-errors/

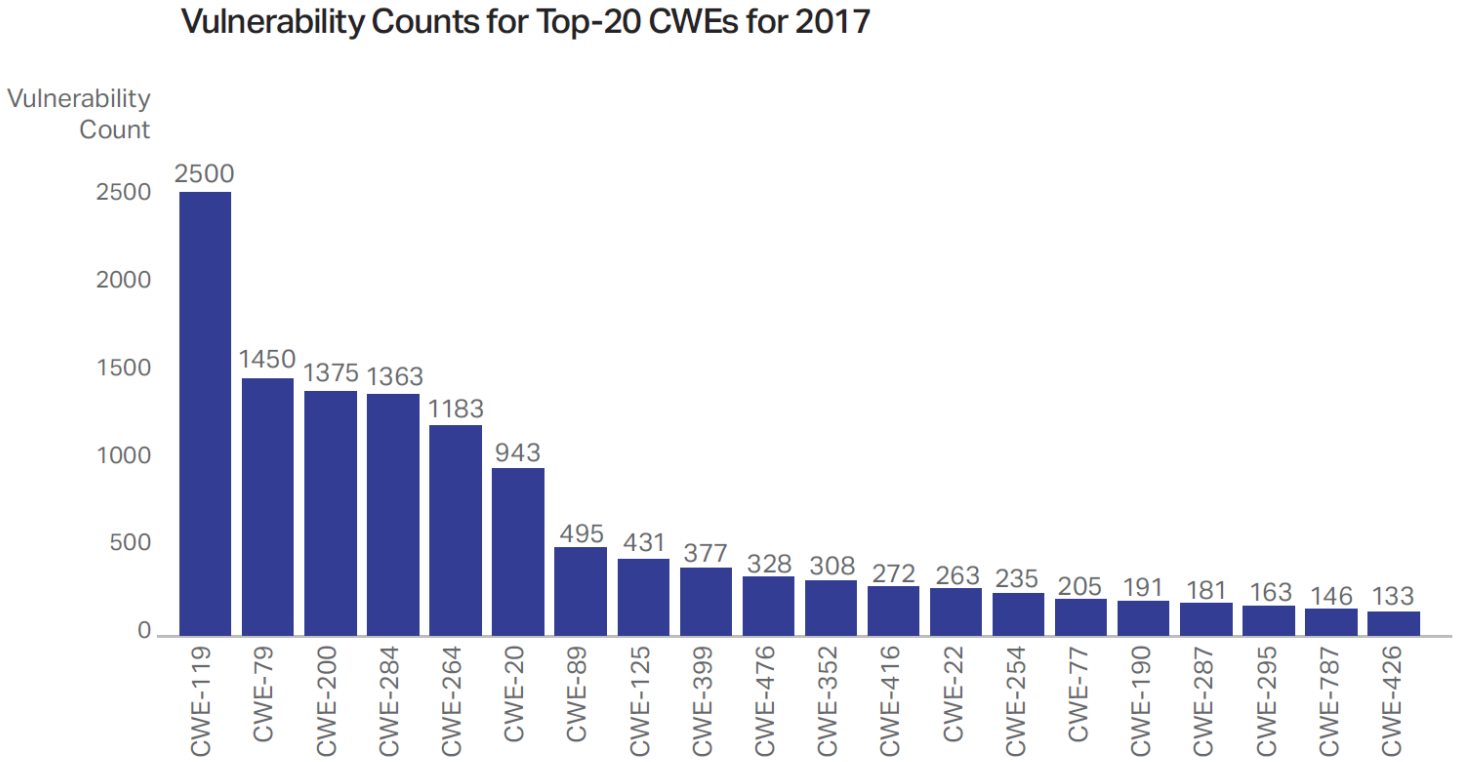
CWE/SANS Top 25 most dangerous software errors

Porous Defenses

- CWE-306 Missing Authentication for Critical Function
- CWE-862 Missing Authorization
- CWE-798 Use of Hard-coded Credentials
- CWE-311 Missing Encryption of Sensitive Data
- CWE-807 Reliance on Untrusted Inputs in a Security Decision
- CWE-250 Execution with Unnecessary Privileges
- CWE-863 Incorrect Authorization
- CWE-732 Incorrect Permission Assignment for Critical Resource
- CWE-327 Use of a Broken or Risky Cryptographic Algorithm
- CWE-307 Improper Restriction of Excessive Authentication Attempts
- CWE-759 Use of a One-Way Hash without a Salt

http://www.sans.org/top25-software-errors/

MicroFocus
2018 Application Security Research Report



LEGEND

CWE-119: (Buffer Overflow)
CWE-79: Cross-site scripting (XSS)
CWE-200: Information exposure
CWE-284: Improper access control
CWE-264: Permissions, privileges, and access control
CWE-20: Improper input validation
CWE-89: SQL injection
CWE-125: Out-of-bounds read
CWE-399: Resource management errors
CWE-476: Null pointer dereference
CWE-352: Cross-site request forgery (CSRF)
CWE-77: Command injection
CWE-190: Integer overflow or wraparound
CWE-287: Improper authentication
CWE-787: Out-of-bounds write
CWE-426: Untrusted search path

Buffer overflow

- A very common attack mechanism
  - first widely used by the Morris Worm in 1988
- Defined in NIST glossary as

  "A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system."
- Prevention techniques known
  - easiest: **use memory safe languages** with automatic input validation!
  - OS, library, and compiler can perform automatic mitigation
- Still of major concern
  - legacy of buggy code in widely deployed operating systems and applications
  - continued careless programming practices by programmers

Buffer overflow basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
  - locations could hold other program variables, parameters, or program control flow data
- Buffer could be located on the stack, in the heap, or in the data section of the process
- To exploit a buffer overflow an attacker needs:
  - to identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
  - to understand how that buffer is stored in memory and determine potential for corruption
- Identifying vulnerable programs can be done by:
  - inspection of program source
  - tracing the execution of programs as they process oversized input
  - using tools such as fuzzing to automatically identify potentially vulnerable programs

Buffer overflow example: code

```
int main(int argc, char *argv[]) {
    int valid = FALSE;
    char str1[8];
    char str2[8];        // because of stack order, str2 will be on lower addresses than str1

    strcpy(str1, "START");
    gets(str2);
    if (strncmp(str1, str2, 8) == 0)
        valid = TRUE;
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
$ cc -fno-stack-protector -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs

Buffer overflow example: stack values

Memory Address	Before gets(str2)		After gets(str2)	Contains Value of
. . . .	. . . .		. . . .	
bffffbf4	34fcffbf 4 . . .		34fcffbf 3 . . .	argv
bffffbf0	01000000		01000000	argc
bffffbec	c6bd0340 . . . @		c6bd0340 . . . @	return addr
bffffbe8	08fcffbf		08fcffbf	old base ptr
bffffbe4	00000000		01000000	valid
bffffbe0	80640140 . d . @		00640140 . d . @	
bffffbdc	54001540 T . . @		4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R		42414449 B A D I	str1[0-3]
bffffbd4	00850408		4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @		42414449 B A D I	str2[0-3]
. . . .	. . . .		. . . .	

Stack buffer overflows

- Occur when buffer is located on stack
  - also referred to as stack smashing
  - used by Morris Worm
  - exploits included an unchecked buffer overflow
- Are still being widely exploited
- Stack frame
  - when one function calls another it needs somewhere to save the return address
  - also needs locations to save the parameters to be passed in to the called function and to possibly save register values

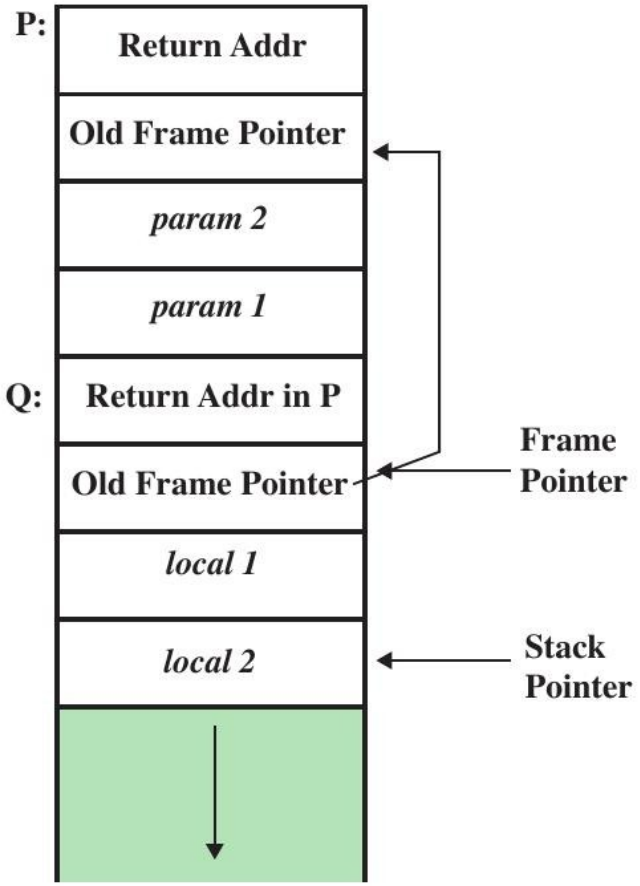


Figure 10.3 Example Stack Frame with Functions P and Q

Common unsafe C standard library routines

`gets(char *str)`	read line from standard input into str
`sprintf(char *str, char *format, ...)`	create str according to supplied format and variables
`strcat(char *dest, char *src)`	append contents of string src to string dest
`strcpy(char *dest, char *src)`	copy contents of string src to string dest
`vsprintf(char *str, char *fmt, va_list ap)`	create str according to supplied format and variables

Table 10.2 Some Common Unsafe C Standard Library Routines

Buffer overflow example: code

```
$ cc -g -o buffer1 buffer1.c
buffer1.c: In function ‘main’:
buffer1.c:10:5: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’?
[-Wimplicit-function-declaration]
   10 |     gets(str2);
      |     ^~~~
      |     fgets
/usr/bin/ld: /tmp/ccQdK5WB.o: in function `main':
buffer1.c:10: Warning: the `gets' function is dangerous and should not be used.

$ ./buffer1
BADINPUTBADINPUT
buffer1: str1(START), str2(BADINPUTBADINPUT), valid(0)
*** stack smashing detected ***: terminated
[1]    1265340 abort (core dumped)  ./buffer1
```

(c) Basic buffer overflow example runs with modern default compiler options

Shellcode

- Code supplied by attacker
  - often saved in buffer being overflowed
  - traditionally transferred control to a user command-line interpreter (shell)
- Machine code
  - specific to processor and operating system
  - traditionally needed good assembly language skills to create
  - more recently a number of sites and tools have been developed that automate this process
- Metasploit project
  - provides useful information to people who perform penetration, IDS signature development, and exploit research
  - see https://www.metasploit.com/

Compile-time defenses: Programming language

- Use a modern high-level language
  - not vulnerable to buffer overflow attacks (but beware of calling native code libraries!)
  - compiler enforces range checks and permissible operations on variables (with some performance penalty)
  - e.g. Rust, Java/Kotlin/Scala, Go, C#/F#, Haskell, ...
- Scripting languages are typically not susceptible to buffer overflow attacks
  - however, dynamic typing has other problems…
  - e.g. Python, Javascript, Perl, Ruby, ~~PHP~~, …
    - not in language, but runtime, function libraries, etc. may have (had) problems (=bugs)

Compile-time defenses: Safe coding techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
  - an example of this is the OpenBSD project
  - OpenBSD programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
  - this has resulted in what is widely regarded as one of the safest operating systems (among those written in C/C++) in active use

Compile-time defenses: Language extensions / libs

- Handling dynamically allocated memory is more problematic because the size information is not available at compile time
  - requires an extension and the use of library routines
    - programs and libraries need to be recompiled
    - likely to have problems with third-party applications
- Concern with C is use of unsafe standard library routines
  - one approach has been to replace these with safer variants
    - `libsafe` is an example
    - library is implemented as a dynamic library arranged to load before the existing standard libraries

Compile-time defenses: Stack protection

- Add function entry and exit code to check stack for signs of corruption
- Use random canary
  - value needs to be unpredictable
  - should be different on different systems
- StackGuard/ProPolice and Return Address Defender (RAD)
  - GCC extensions that include additional function entry and exit code
    - function entry writes a copy of the return address to a safe region of memory
    - function exit code checks the return address in the stack frame against the saved copy
    - if change is found, aborts the program
  - enable with `-fstack-protector-strong` or `-fstack-protector-all`
- AddressSanitizer in Clang/LLVM and newer GCC
  - also detects other errors, e.g. use-after-free → **turn on by default!**
  - enable with `-fsanitize=address` and `-fsanitize=bounds`

Buffer overflow example: code

```
$ cc -fsanitize=address -fsanitize=bounds -fstack-protector-all -g -o buffer1 buffer1.c
<same compile-time warnings as before>
$ ./buffer1
BADINPUTBADINPUT
=================================================================
==1270147==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffd11a17cf8 at pc
0x7f6139cdfdbb bp 0x7ffd11a17b40 sp 0x7ffd11a172b8
READ of size 17 at 0x7ffd11a17cf8 thread T0
    #0 0x7f6139cdfdba  (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x9cdba)
    #1 0x7f6139ce0ddc in __interceptor_vprintf (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x9dddc)
    #2 0x7f6139ce0ed6 in printf (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x9ded6)
    #3 0x5567e6afc38e in main buffer1.c:13
    #4 0x7f613910b0b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x270b2)
    #5 0x5567e6afc1ad in _start (buffer1+0x11ad)

Address 0x7ffd11a17cf8 is located in stack of thread T0 at offset 72 in frame
    #0 0x5567e6afc278 in main buffer1.c:4

  This frame has 2 object(s):
    [32, 40) 'str1' (line 6)
    [64, 72) 'str2' (line 7) <== Memory access at offset 72 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism,
swapcontext or vfork (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x9cdba)
...
```

Run-time defenses: Data Execution Prevention (DEP)

- Prevent execution in data memory pages
- Modes
  - hardware: CPU checks NX/XD/XN bit of page
    - blocks execution of code in page
    - AMD64 (Athlon 64, Opteron), Intel from Pentium 4, modern ARM CPUs
  - software
- OS support
  - Linux (2000), Windows XP SP2 (2004), Mac OS X (2006), ...
- Limitations
  - no protection against “return to libc” attack
  - may break legitimate uses (JIT-Compiler)
  - program compatibility

Run-time defenses: Data Execution Prevention (DEP)

- POSIX
  - page access permissions
  - PROT_READ, PROT_WRITE, PROT_EXEC
- OpenBSD / Mac OS X
  - W^X: Write XOR Execute
  - hardware and emulation
- Linux
  - ExecShield (patch)
    - hardware and emulation
    - ASCII armor region: uses addresses from 0 to 0x01010100
  - PaX (patch)
    - hardware and emulation
    - ASLR (see next slide)

Run-time defenses: Address space randomization

Address space layout randomization (ASLR)

- Manipulate location of key data structures
  - stack, heap, global data
  - using random shift for each process
  - large address range (64 bit) on modern systems means wasting some has negligible impact
  - but: on 32 bit architectures not enough entropy for sufficient protection against brute force address tries
- Randomize location of heap buffers
- Random location of standard library functions
- Implementations
  - virtual memory, PIE (position-independent executable)
  - Linux (getting stronger over time, including KASLR for kernel memory)
  - Windows (since Vista), Mac OS X (weak), iOS

Run-time defenses: Guard pages

- Place guard pages between critical regions of memory
  - flagged in MMU as illegal addresses
  - any attempted access aborts process
  - NOP slides: Lots of No-Op commands with actual code at end. If you land somewhere, you will execute the code → likely to hit guard page
    - specific attacks may only be 100 bytes long → guard page not very useful
- Further extension places guard pages between stack frames and heap buffers
  - cost in execution time to support the large number of page mappings necessary
- Beginning to be supported by hardware, e.g. ARM Memory Tagging (MTE)

Variants of buffer overflow attacks

- Replacement stack frame:
  - putting “fake” new stack frame into overwritten buffer and overwriting frame pointer address
  - dummy stack frame contains new return address to shellcode
  - function returns normally (original return address is not changed), but then calling function uses dummy stack frame and jumps to shellcode when itself returns
  - may allow circumventing run-time checks on return code
  - variant: off-by-one attack
- Return to system call: see next slide
- Heap overflow: even more indirect to work around stack protections
- Global data area overflow: see next slides
- Others

Return to system call

Stack overflow variant replaces return address with standard library function

- Response to non-executable stack defenses
- Attacker constructs suitable parameters on stack above return address
- Function returns and library function executes
- Attacker may need exact buffer address
- Can even chain two or more library calls

Defenses

- Any stack protection mechanisms to detect modifications to the stack frame or return address by function exit code
- Use non-executable stacks
- Randomization of the stack in memory and of system libraries

Global data overflow

Can attack buffer located in global data

- May be located above program code
- If it has function pointer and vulnerable buffer
- Or adjacent process management tables
- Aim to overwrite function pointer later called

Defenses

- Non executable or random global data region
- Move function pointers
- Guard pages

Software security, quality, and reliability

Software quality and reliability

- Concerned with the *accidental* failure of program as a result of some theoretically random, unanticipated input, system interaction, or use of incorrect code
- Improve using structured design and testing to identify and eliminate as many bugs as possible from a program
- Concern is not how many bugs, but how often they are triggered

Software security

- *Attacker chooses probability distribution*, specifically targeting bugs that result in a failure that can be exploited by the attacker
- Triggered by inputs that differ dramatically from what is usually expected
- Unlikely to be identified by common testing approaches
- **Software should only do what it is intended to, do it timely, and nothing else**

Defensive programming

Problem with current practices

- Programmers often make assumptions about the type of inputs a program will receive and the environment it executes in
  - assumptions need to be validated by the program and all potential failures handled gracefully and safely
- Requires a changed mindset to traditional programming practices
  - programmers have to understand how failures can occur and the steps needed to reduce the chance of them occurring in their programs

Defensive programming

- A form of defensive design to ensure continued function of software despite unforeseen usage
- Requires attention to all aspects of program execution, environment, and type of data it processes
- Also called secure programming
- **Assume nothing, check all potential errors**
  - programmer never assumes a particular function call or library will work as advertised so handles it in the code

Security by design

- Security and reliability are common design goals in most engineering disciplines
- Software development not as mature
  - much higher failure levels tolerated
- Despite having a number of software development and quality standards
  - main focus is general development lifecycle
  - increasingly identify security as a key goal
- **Don't:**
  - trust user or network input
  - trust external systems
  - trust infrastructure
  - mix code and data
  - store any data you don't need (temporarily or permanently)

Root/admin privileges in software

antivirus and other security add-ons often run as admin

- Programs with root / administrator privileges are a major target of attackers
  - they provide highest levels of system access and control
  - are needed to manage access to protected system resources
- Often privilege is only needed at start (e.g. to bind to privileged network port or open key files)
  - can then drop privileges and run as normal/limited user
- Good design partitions complex programs in smaller modules with needed privileges → **isolation/compartmentalization** design
  - provides a greater degree of isolation between the components
  - reduces the consequences of a security breach in one component
  - easier to test and verify

Input size validation

- Programmers often make assumptions about the maximum expected size of input
  - allocated buffer size is not confirmed
  - resulting in buffer overflow
- Many other input parsing problems in addition to (trivial) size overflow issues exist, especially with complex formats
- Testing may not identify vulnerability
  - test inputs are unlikely to include large enough and/or complex enough inputs to trigger the overflow / parsing error
  - use **fuzzing**!
- Safe coding treats all input as dangerous

Interpretation of program input

- Program input may be binary or text
  - binary interpretation depends on encoding and is usually application specific
- There is an increasing variety of character sets being used
  - care is needed to identify just which set is being used and what characters are being read
- Failure to validate may result in an exploitable vulnerability

Injection attacks

… are flaws relating to invalid handling of input data, specifically when program input data can accidentally or deliberately influence the flow of execution of the program

- Very problematic for interpreted scripting languages (e.g. PHP) where direct code injection attack is possible
- On client side one of the biggest attack vectors (e.g. PDF)
- Common type of server side attack: SQL injection attack
  - user supplied input is used to construct a SQL request to retrieve information from a database
  - vulnerability is similar to command injection
    - difference is that SQL metacharacters are used rather than shell metacharacters
  - to prevent this type of attack the input must be validated before use
- Common type of web attack: cross site scripting (XSS) attack
  - user supplied content (e.g. from cookie) included in web page as displayed to other users and executed in their browsers

Race conditions

- Without synchronization of accesses it is possible that values may be corrupted or changes lost due to overlapping access, use, and replacement of shared values
- Arise when writing concurrent code whose solution requires the correct selection and use of appropriate synchronization primitives
- Deadlock
  - processes or threads wait on a resource held by the other
  - one or more programs has to be terminated
- In practice, often a problem with temporary files
  - application (tries to) create temporary file (possibly with root access)
  - attacker creates the file, but with different permissions/ownership/link target
  - application then writes into the file created by attacker
    → possibly writes into different target with elevated privileges

Preventing race conditions

… is hard (compare to multi-threaded programming issues)

- Need suitable synchronization mechanisms
  - most common technique is to acquire a lock on the shared file
- Lockfile
  - process must create and own the lockfile in order to gain access to the shared resource
  - concerns
    - if a program chooses to ignore the existence of the lockfile and access the shared resource the system will not prevent this
    - all programs using this form of synchronization must cooperate
    - implementation

Safe temporary files

- Many programs use temporary files
- Often in common, shared system area
- Must be unique, not accessed by others
- Commonly create name using process ID
  - unique, but predictable
  - attacker might guess and attempt to create own file between program checking and creating
- Secure temporary file creation and use requires the use of random names
  - better: **use OS function** to create unique randomly named file

Input fuzzing

- Developed by Barton Miller at the University of Wisconsin Madison in 1989
- Software testing technique that uses randomly generated data as inputs to a program
  - range of inputs is very large
  - intent is to determine if the program or function correctly handles abnormal inputs
  - simple, free of assumptions, cheap
  - assists with reliability as well as security
- Can also use templates to generate classes of known problem inputs
  - disadvantage is that bugs triggered by other forms of input would be missed
  - combination of approaches is needed for reasonably comprehensive coverage of the inputs
  - difficulty: how to detect problem from output

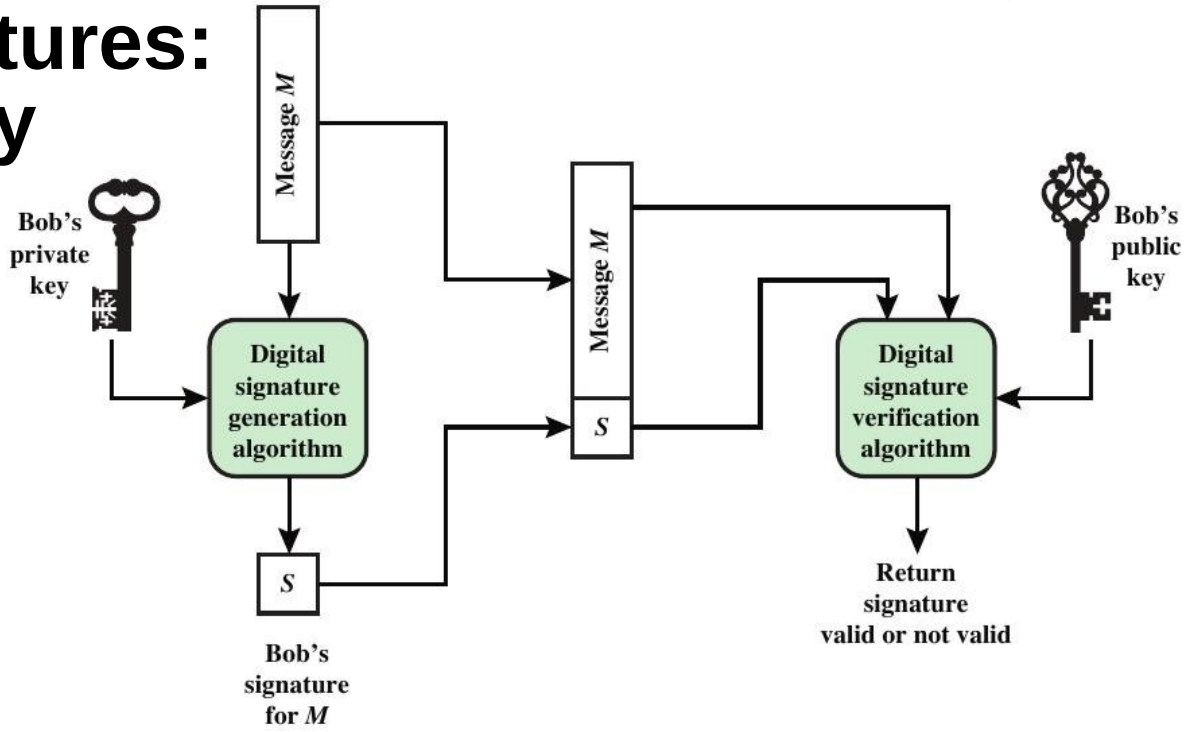
Handling program output

- Final component is program output
  - may be stored for future use, sent over network, or displayed
  - may be binary or text
- Important from a program security perspective that the output conform to the expected form and interpretation
- Programs must identify what is permissible output content and filter any possibly untrusted data to ensure that only valid output is displayed
- Character set should be specified

Software signatures

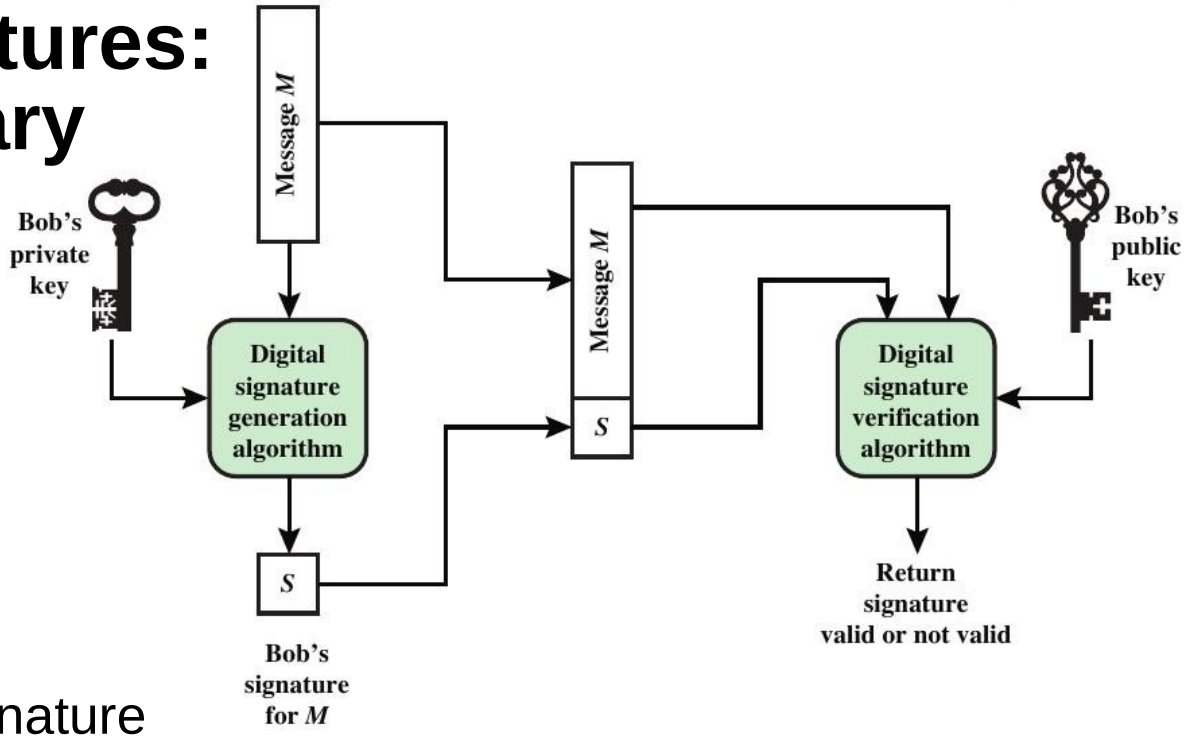
- (Stored or transmitted) code itself can become the target of attacks
  - e.g. virus modifying other code
  - e.g. malware being inserted into otherwise benevolent code in transit
- This is an attack against the integrity of the code
  - have a standard cryptographic method to protect against integrity violation: digital signatures
  - since code is rarely transmitted in a mutually authenticated secure channel, typically use asymmetric (and not symmetric) signatures
- Different components required for code signatures
  - cryptographic algorithms and packet/executable formats → easy
  - key management of private key at developer side → ideally offline
  - unspoofable/authentic public key distribution to all verifying instances → **this is the hard problem**

Software signatures: signing a binary



- Apply standard asymmetric signature
  - hash program binary ("the code")
  - apply RSA or ECDSA (in the future PQC singature) with private key
  - attach meta data (e.g. identity of signer) and signature to code (careful not to modify the binary in this process and thus invalidate signature → required package standard with added signatures)

Software signatures: verifying a binary



- Verify asymmetric signature
  - extract signature value from package format
  - hash program binary ("the code")
  - apply RSA or ECDSA verification with public key
  - **main problem:** how to receive and authenticate public key of developer
    sub problem: how to identify real developer
  - often involves certificate authority (identification of developer still problematic)

Software signatures: distributing public keys

- One (e.g. OS) vendor can ship public keys for verifying additional components with the software package
  - works for drivers, add-ons, and other modules by the same vendor
  - works if that vendor also re-signs and re-distributes third-party code (e.g. Microsoft for Windows drivers)
- One vendor can run its own CA
  - can sign public keys of (verified) developers
  - developers then sign their own code and attach their certificate in addition to the signature
  - verifying code uses CA public key (which must be shipped e.g. with the OS) to first verify the certificate and then, with the public key contained in the certificate, the code
  - works if all developers register with one vendor (e.g. Apple)
- Every developer can create their own keypair/CA
  - no single point of failure (or censorship)
  - but public keys not necessarily authentic → rely on key continuity concepts
  - e.g. Android apps

Deterministic/reproducible/auditable builds

Open issue: does the binary correspond to the source?

- Issue is ignored by most programmers
  - assumption is that the compiler or interpreter generates or executes code that validly implements the language statements
  - additional assumption is that the compiler/library/kernel/hardware itself is not malicious (cf. [Ken Thompson: “Reflections on Trusting Trust”, Communication of the ACM, Vol. 27, No. 8, August 1984, pp. 761-763], online at http://cm.bell-labs.com/who/ken/trust.html)
- Requires comparing machine code with original source
  - slow and difficult
- Development of computer systems with very high assurance level is the one area where this level of checking is required
  - specifically Common Criteria assurance level of EAL 7
- Starting to become a practical possibility
  - Gitian with multiple builders (http://gitian.org/) used by Bitcoin client and Tor browser bundle (https://blog.torproject.org/blog/deterministic-builds-part-two-technical-details)
  - Debian aims at reproducible builds for its packages (https://wiki.debian.org/ReproducibleBuilds): 61% (of 21448 packages) reproducible on 2014-11-11, 22462/24351 (92.2%) on 2016-12-12, 28893/30363 (95.1%) on 2021-01-01
  - Android reproducibility reports: https://android.ins.jku.at/reproducible-builds/
  - if you are looking for a Master's thesis topic, this *still* is one :-)