SJSU SAN JOSÉ STATE UNIVERSITY

Lesson 9 – Software Flaws

Yan Chen

CS166 Fall 2024

- A signed message may be costly to compute & send
  - Since signee needs to send both S = $[M]_{signee}$ and M
- Hash function: “map” big M to smaller “fingerprint” of M
  - Notation: h(M), also called hash, or digest
  - Collisions (different M maps to same hash) exist since input space is larger than output space
- Birthday problem is used to understand collisions
- If find a collision, hash is broken
  - If h(M) has n bits, 2^n different hashes total, 2^n comparisons, and need $2^{n/2}$ tries to break it (find a collision) by “brute-force”

- Properties of a secure crypto hash
  - Deterministic, compressive, efficient, one-way, avalanche effect, weak collision resistance, strong collision resistance
  - Lots of collisions exist, but hard to find any
- Non-crypto hash examples
  - $h(X) = [nX_1 + (n - 1)X_2 + (n - 2)X_3 + \ldots + X_n] \bmod 256$ is used in a non-crypto application rsync
  - Cyclic Redundancy Check (CRC) used for detecting burst errors (but also has been mistakenly used where crypto integrity check is required)

Crypto Hash Functions

- Crypto hash functions similar to block ciphers
  - ➢ The message is hashed in blocks
  - ➢ The hash function consists of some number of rounds
- MD5 (Message-Digest algorithm) & SHA-1 (Secure Hash Algorithm) are popular hash functions
  - ➢ Both broken now, so not used for encryption any more
  - ➢ But still widely used for integrity or other non-crypto apps
- Tiger Hash: "fast & strong"
  - ➢ Optimized for 64-bit processors
  - ➢ Can be replacement for MD5 or SHA-1

- HMAC: hashed MAC, used for integrity
- Online bids: bidders submit h(bid) instead of bid
  - Hashes don’t reveal bids (one way)
  - Can’t change bid after hash sent (collision)
- Reduce spam email: request sender to prove they did some “work” (“proof-of-work”) before accepting email
  - Make spam more costly to send to limit the amount
  - Sender needs to compute 2^N hashes to find a required value
  - Recipient only needs to hash 1 time to verify
  - Acceptable for normal email, but too high for spammers

- All security features are implemented in software
  - ➢ If software is subject to attack, security can be broken
  - ➢ Regardless of strength of crypto, access control, or protocols
- Unfortunately, software is a poor foundation for security
  - ➢ "Bad" software are anywhere...
  - ➢ E.g., NASA Mars Lander, Denver airport, etc.
- Trudy takes advantage of bad software

Alice & Bob	Trudy
Find bugs and flaws by accident	Actively looks for bugs and flaws
Hate bad software…	Likes bad software…
…but they learn to live with it	…and tries to make it misbehave
Must make bad software work	Attacks systems via bad software

- Will focus on insecurity in software
- Lesson 9 – Program flaws (unintentional)
  - ➢ Buffer overflow, incomplete mediation, race conditions
- Lesson 10 – Malicious software (intentional)
  - ➢ Timeline of well-known malware
  - ➢ Ways to detect malware
  - ➢ How advanced malware can evade detection
- Lesson 11 – Software Reverse Engineering (SRE)
- Lesson 12 – Miscellaneous attacks on software
  - ➢ Salami attack, linearization attack, time bomb

- Secure software engineering requires that software does what is intended…
  - …and nothing more
  - Absolutely secure software? Dream on…
  - Absolute security anywhere is impossible
- Program flaws are unintentional
  - But can still create security risks
  - Will cover the common ones...
- "Complexity is the enemy of security"
  - By Paul Kocher, Cryptography Research, Inc.

- A real-world OS system can have millions of lines of code (LOC)!
  - E.g.: Windows XP, 40M LOC; Mac OS X 10.4, 86M LOC; etc.
- Suppose: ~5 bugs per 10K LOC (K = thousands)
  - If a software has 100K LOC, then ~50 bugs per software
  - If a computer has 3K software, then ~150K bugs per computer
  - So, 30K-node network has ~4.5 billion bugs!
- Suppose only 10% (450 million) are security-critical
  - And 10% of security-critical bugs are remotely exploitable
  - Then "only" 45 million critical security flaws!

- An **error** is a programming mistake
  - To err is human
- An error may lead to incorrect state: **fault**
  - A fault is internal to the program
- A **fault** may lead to a failure: system behaves incorrectly
  - A failure is externally observable
- Example:

```
char array[10];
for(i = 0; i < 10; ++i)
        array[i] = 'A';
array[10] = 'B';
```

- We use the term flaw for all of the above

- Buffer overflow: data is larger than the memory space ("buffer") that's allocated
- What happens when the following C code is executed?

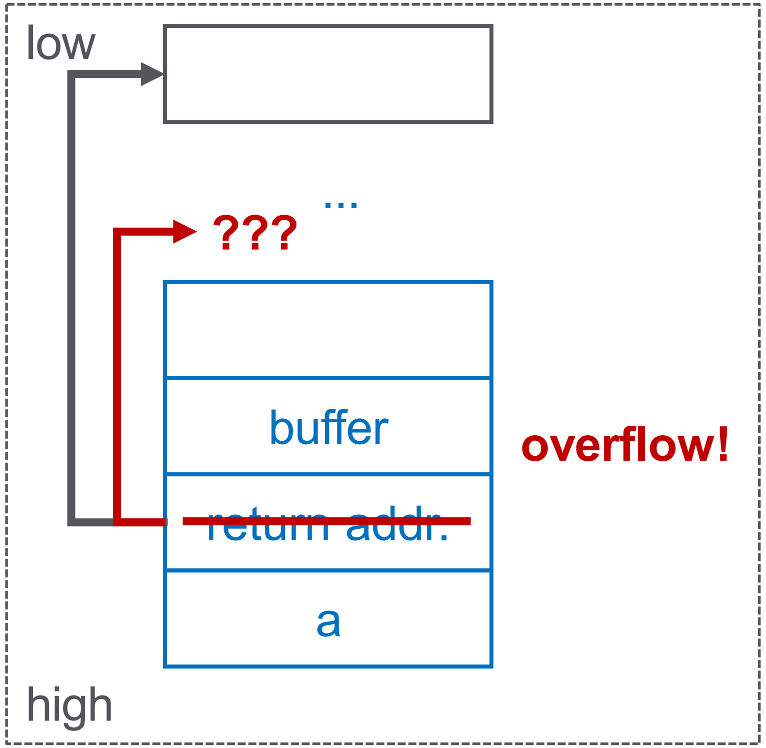
```
int main(){
   int buffer[10];
   buffer[20] = 37;
}
```

  - Depending on what resides in memory at location "buffer[20]"
  - Might overwrite user or system data or code!
  - Or program could work just fine
- Simple example: boolean flag for authentication
  - Buffer overflow could overwrite flag so everyone got accepted

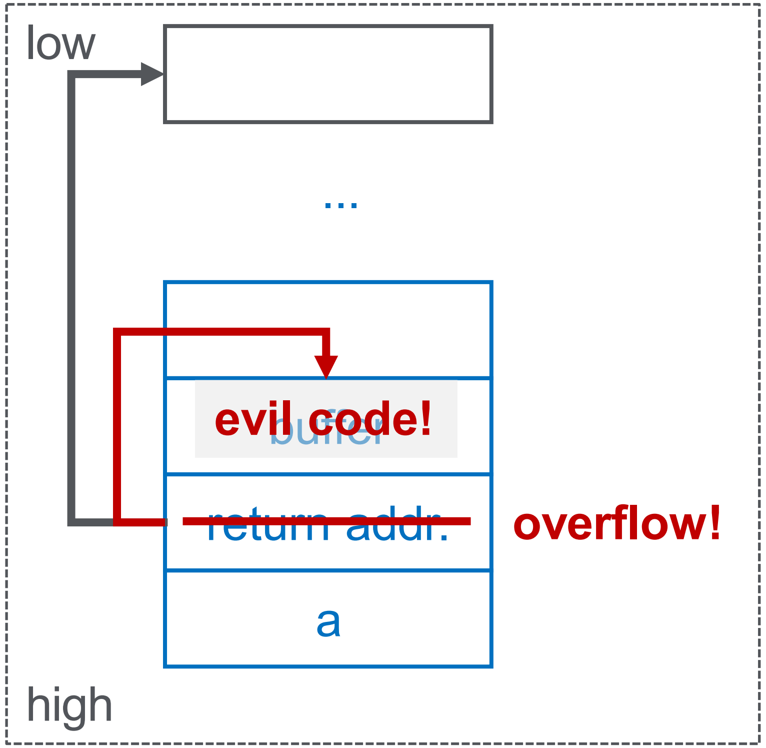
- Consider a simplified memory organization...
  - From low to high: code, static variables, dynamic data, stack
  - Stack is used as a "scratch paper" for dynamic local variables, parameters to functions, and return address
- Consider the following code

```
void func(int a){
   char buffer[10];
}
void main(){
   func(1);
}
```

  - If buffer overflows, program will "return" to wrong location!



- Trudy has a better idea…
  - Code injection!
  - First, direct the return address to the start of buffer
  - Then, fill the buffer with her executable “evil code”
  - I.e., Trudy can run code of her choosing…on your machine!
- Need some trial-and-error to find the addresses
  - Start of buffer
  - And return address



- To do a buffer overflow attack...
  - A buffer overflow must exist in the code
- Not all buffer overflows are exploitable
  - Things must align properly
- If exploitable, attacker can inject code
  - Trial and error is likely required though
  - Lots of help is available online...
- Stack smashing is “attack of the decade”…
  - …for many recent decades
  - Also heap & integer overflows, format strings, etc.

- Several ways to defense stack smashing
  - Non-executable stack, **canary**, **ASLR**
  - Use safe languages (Java, C#)
  - Use safer C functions
- Canary: Run-time stack check
  - Push canary onto stack
  - Set canary value to constant 0x000aff0d (or depends on ret)
  - If canary value is overridden, then there is overflow!
  - E.g.: Microsoft allows user to define a handler function called when canary died (but handler can be specified by attacker!)

- ASLR: Address Space Layout Randomization
  - Randomize place where code loaded in memory
  - Makes most buffer overflow attacks probabilistic
  - e.g., Windows Vista uses 256 random layouts, so about 1/256 chance buffer overflow works
  - Similar thing in Mac OS X and other OSs
  - However, Attacks against Microsoft's ASLR do exist
  - Possible to "de-randomize"

- Incomplete mediation: software not validating user input
  - ➢ Can result in buffer overflow attacks, web attacks, etc.
- This is a common mistake...
  - ➢ Even in Linux kernel, which is consider as a “good” software since it’s open source, and written by experienced people...
  - ➢ Lots of buffer overflows in Linux due to incomplete mediation!
- Example: consider `strcpy(buffer, argv[1])`
  - ➢ Suppose a buffer overflow occurs if `len(buffer) < len(argv[1])`
  - ➢ Software must validate the input by checking the length of `argv[1]`, otherwise, incomplete mediation

A Subtle Example

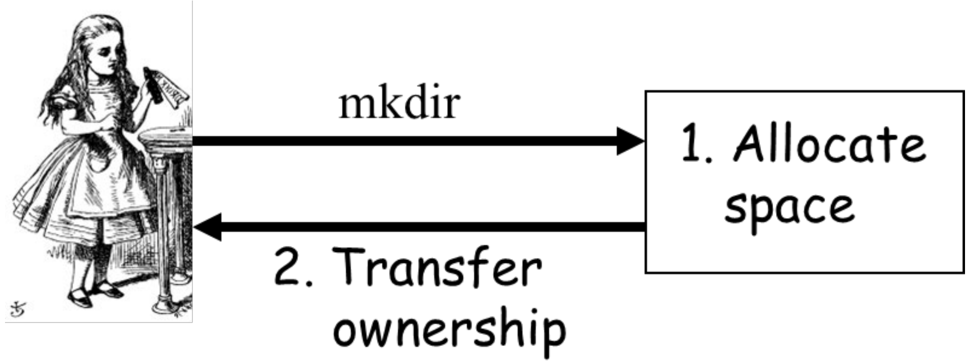
- A subtle example: data that is input to a Web form
  - Suppose input is validated on client
  - For example, the following is valid:

    http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=205
  - But if input is not checked on server...
  - Then attacker could send http message

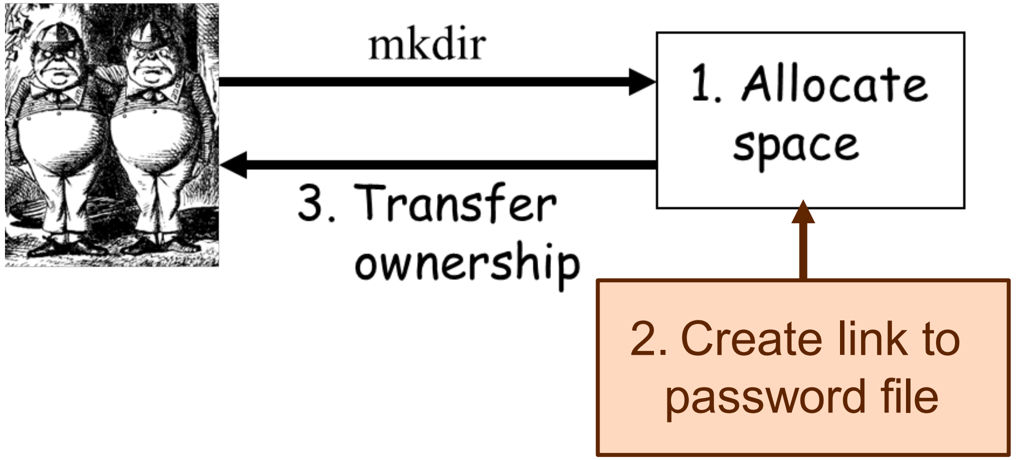
    http://www.things.com/orders/final&custID=112&num=55A&qty=20&price=10&shipping=5&total=25
  - That is, validation on client only is NOT enough!

- Race conditions can arise when security-critical process occurs in stages
  - Attacker makes change between stages
  - “Race" between the attacker and the next stage of the process
  - Often, between stage that gives authorization
  - But before stage that transfers ownership
- Race conditions are common, but harder to exploit
- To prevent, make security-critical processes “atomic”
  - Occur all at once, not in stages
  - Not always easy to accomplish in practice though...

- Example: Unix `mkdir` (old version)
  - `mkdir` creates new directory



  - Possible attack (timing is important…):



- Malware (intentionally “bad” software)
  - Timeline of well-known malware
  - Detect malware
  - Malware try to evade detection

Concepts

Exercises

- Secure software
- Flaws
  - Error, fault, failure
- Buffer overflow
  - Defenses: canary, ASLR
- Incomplete mediation
- Race condition

- In contrast to the stack-based buffer overflow discussed
  - Explain how a heap-based buffer overflow works
  - Explain how an integer overflow works
- Discuss an example of a real-world race condition, other than the `mkdir` example presented

References

- Stamp, Mark, “Information Security, Principles and Practice, 2nd ed.,” Wiley, New Jersey, USA, 2011