CISC 3325 - Information Security

Data Integrity, Programs and Programming

Topics for today

- Data Integrity
- Programs and programming

Data Integrity:
Applications of Cryptographic Hash Functions

Error Detecting Codes

- Demonstrates that a block of data has been modified
- Simple error detecting codes:
  - Parity checks
    - Parity bit added to a string of binary code to ensure that the total number of 1-bits in the string is even or odd
  - Cyclic redundancy checks – used on hardware devices
- Cryptographic error detecting codes:
  - One-way hash functions
  - Cryptographic checksums
  - Digital signatures

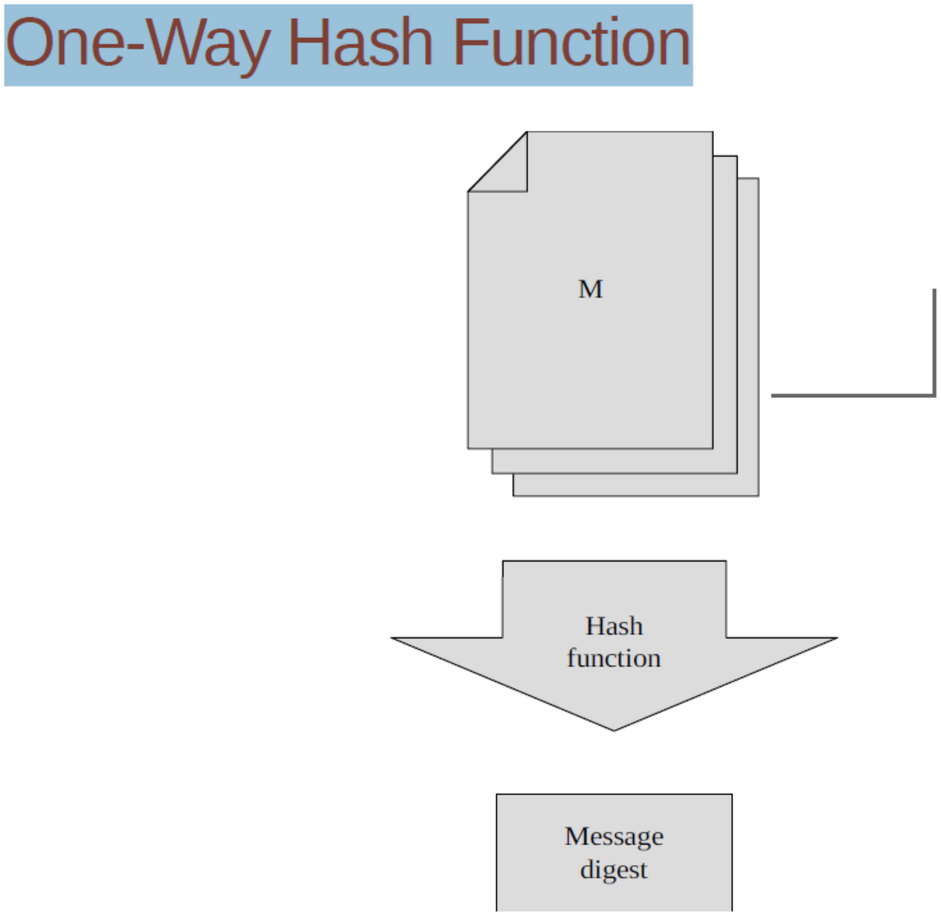
Parity Check

Original Data	Parity Bit	Modified Data	Modification Detected?
0 0 0 0 0 0 0 0	1	0 0 0 0 0 0 0 1	
0 0 0 0 0 0 0 0	1	1 0 0 0 0 0 0 0	
0 0 0 0 0 0 0 0	1	1 0 0 0 0 0 0 1	
0 0 0 0 0 0 0 0	1	0 0 0 0 0 0 1 1	
0 0 0 0 0 0 0 0	1	0 0 0 0 0 1 1 1	
0 0 0 0 0 0 0 0	1	0 0 0 0 1 1 1 1	
0 0 0 0 0 0 0 0	1	0 1 0 1 0 1 0 1	

Parity Check

Original Data	Parity Bit	Modified Data	Modification Detected?
0 0 0 0 0 0 0 0	1	0 0 0 0 0 0 0 1	Yes
0 0 0 0 0 0 0 0	1	1 0 0 0 0 0 0 0	Yes
0 0 0 0 0 0 0 0	1	1 0 0 0 0 0 0 1	No
0 0 0 0 0 0 0 0	1	0 0 0 0 0 0 1 1	No
0 0 0 0 0 0 0 0	1	0 0 0 0 0 1 1 1	Yes
0 0 0 0 0 0 0 0	1	0 0 0 0 1 1 1 1	No
0 0 0 0 0 0 0 0	1	0 1 0 1 0 1 0 1	No

One-Way Hash Function



Adapted From *Security in Computing, Fifth Edition*, by Charles P. Pfleeger, et al. (ISBN: 9780134085043). Copyright 2015 by Pearson Education, Inc.

Hash Functions

- A hash function h maps a plaintext x to a fixed-length value x = h(P) called hash value or digest of P
  - A collision is a pair of plaintexts P and Q that map to the same hash value, h(P) = h(Q)
  - Collisions are unavoidable
  - For efficiency, the computation of the hash function should take time proportional to the length of the input plaintext

Cryptographic Hash Functions

- A cryptographic hash function satisfies additional properties
  - Preimage resistance (aka one-way)
    - Given a hash value x, it is hard to find a plaintext P such that h(P) = x
  - Second preimage resistance (aka weak collision resistance)
    - Given a plaintext P, it is hard to find a plaintext Q such that h(Q) = h(P)
  - Collision resistance (aka strong collision resistance)
    - It is hard to find a pair of plaintexts P and Q such that h(Q) = h(P)
- Collision resistance implies second preimage resistance
- Hash values of at least 256 bits recommended to defend against brute-force attacks

Birthday Attack

- The brute-force birthday attack aims at finding a collision for a hash function h
  - Randomly generate a sequence of plaintexts X1, X2, X3,…
  - For each Xi compute yi = h(Xi) and test whether yi = yj for some j < i
  - Stop as soon as a collision has been found
- If there are m possible hash values, the probability that the i-th plaintext does not collide with any of the previous i -1 plaintexts is 1 - (i - 1)/m

Birthday Attack

- Probability F_k that the attack fails (no collisions) after k plaintexts is

$$F_k = (1 - 1/m)\,(1 - 2/m)\,(1 - 3/m)\,\ldots\,(1 - (k - 1)/m)$$

- Using the standard approximation $1 - x \approx e^{-x}$

$$F_k \approx e^{-(1/m\, +\, 2/m\, +\, 3/m\, +\, \ldots\, +\, (k-1)/m)} = e^{-k(k-1)/2m}$$

- The attack succeeds/fails with probability ½ when $F_k = ½$, that is,

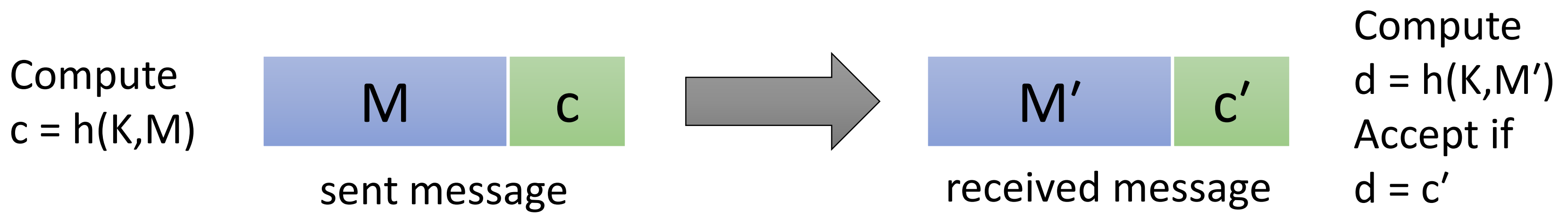
$$e^{-k(k-1)/2m} = ½$$

$$k \approx 1.17\ m^{½}$$

  - We conclude that a hash function with b-bit values provides about b/2 bits of security

Message Authentication Code (MAC)

- Cryptographic hash function h(K,M) with two inputs:
  - Secret key K
  - Message M
- Message integrity with MAC
  - Sequence of messages transmitted over insecure channel
  - Secret key K shared by sender and recipient
  - Sender computes MAC c = h(K,M) and transmits it along with message M
  - Receiver recomputes MAC from received message and compares it with received MAC
  - Attacker cannot compute correct MAC for a forged message
  - More efficient than signing each message
  - Secret key can be sent in a separate encrypted and signed message

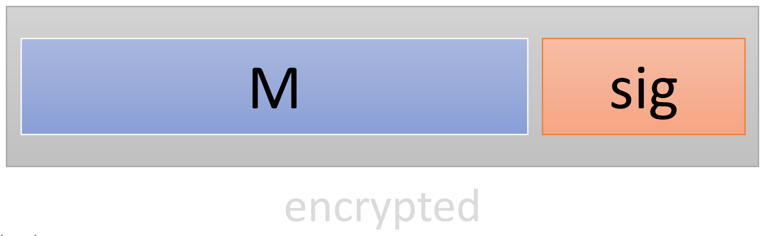


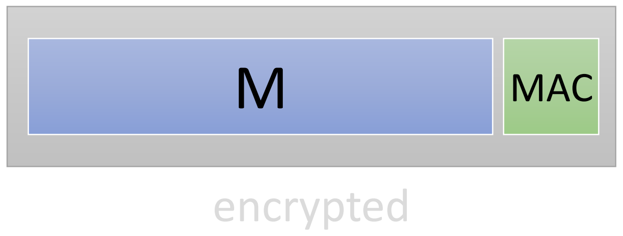
HMAC

- Building a MAC from a cryptographic hash function is not immediate
- Because of the iterative construction of standard hash functions, the following MAC constructions are insecure:
  - $h(K \| M)$
  - $h(M \| K)$
  - $h(K \| M \| K)$
- HMAC provides a secure construction:
  - $h(K \oplus A \| h(K \oplus B \| M))$
  - A and B are constants
  - Internet standard used, e.g., in IPSEC
  - HMAC security is the same as that of the underlying cryptographic hash function

Securing a Communication Channel

- Assuring both integrity and confidentiality of messages transmitted over an insecure channel
- Sign and encrypt
  - The encrypted pair (message, signature) is transmitted
- MAC and encrypt
  - The encrypted pair (message, MAC) is transmitted
  - Secret key for MAC can be sent in separate message
  - More efficient than sign and encrypt
  - MAC is shorter and faster to compute than signature and verification
- Alternatively, signing or applying MAC could be done on encrypted message

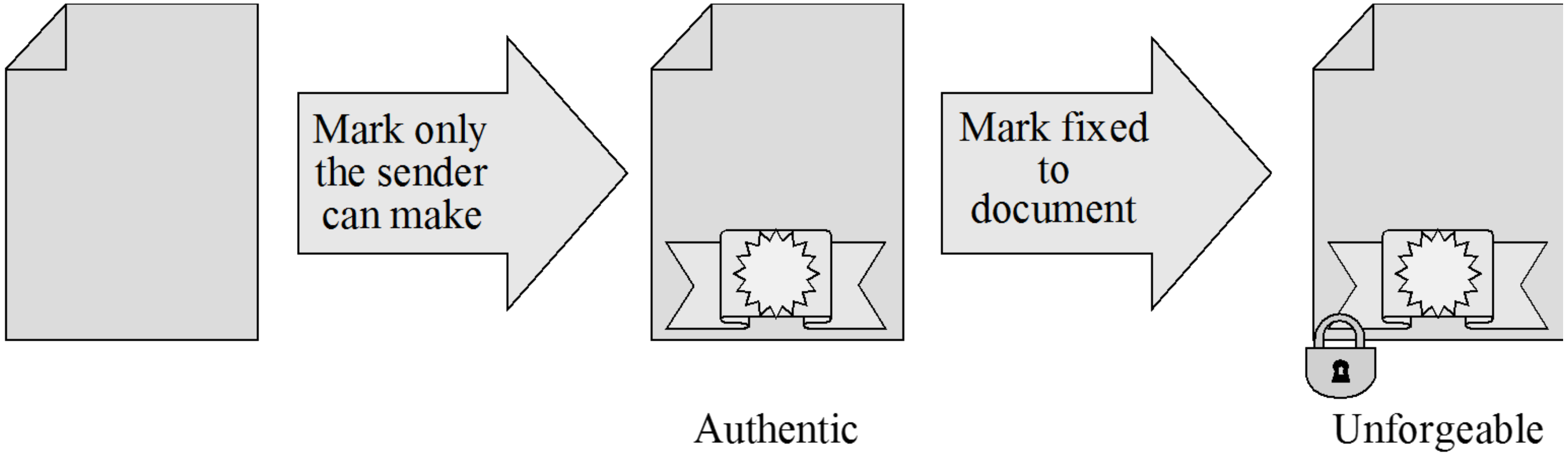




Characteristics of Digital Signatures

- The digital signature is unforgeable.
  - Only person S can create Sig(S,M)
    - where M is the original message.
- It must be authentic
  - The recipient of Sig(S,M) can check that the message is from S.
- The message M can not be changed
  - by R or a third party.
- The message can not be reused.

Digital Signature



How to Implement a Digital Signature

- Use public key encryption:
  - S encrypts the signature (or message) with their private key.
  - R decrypts the message using S's public key.
- Problems:
  - Was R the correct recipient?
  - Is S who they claim to be?
  - Can S trust R and vice versa?

Certificates: Trustable Identities and Public Keys

- A certificate is a public key and an identity bound together and signed by a certificate authority.
- A certificate authority is an authority that users trust to accurately verify identities before generating certificates that bind those identities to keys.
- Certificates have a lifetime and must be maintained.

Digital Certificate Authorities, April, 2016 - Wikipedia

Rank	Issuer	Usage	Market Share
1	Comodo	8.1%	40.6%
2	Symantec	5.2%	26.0%
3	GoDaddy	2.4%	11.8%
4	GlobalSign	1.9%	9.7%
5	IdenTrust	0.7%	3.5%
6	DigiCert	0.6%	3.0%
7	StartCom	0.4%	2.1%
8	Entrust	0.1%	0.7%
9	Trustwave	0.1%	0.5%
10	Verizon	0.1%	0.5%

Certificate Signing and Hierarchy

To create Diana's certificate:

Diana creates and delivers to Edward:

Name: Diana Position: Division Manager Public key: 17EF83CA ...

Edward adds:

Name: Diana Position: Division Manager Public key: 17EF83CA ...	hash value 128C4

Edward signs with his private key:

Name: Diana Position: Division Manager Public key: 17EF83CA ...	hash value 128C4

Which is Diana's certificate.

To create Delwyn's certificate:

Delwyn creates and delivers to Diana:

Name: Delwyn Position: Dept Manager Public key: 3AB3882C ...

Diana adds:

Name: Delwyn Position: Dept Manager Public key: 3AB3882C ...	hash value 48CFA

Diana signs with her private key:

Name: Delwyn Position: Dept Manager Public key: 3AB3882C ...	hash value 48CFA

And appends her certificate:

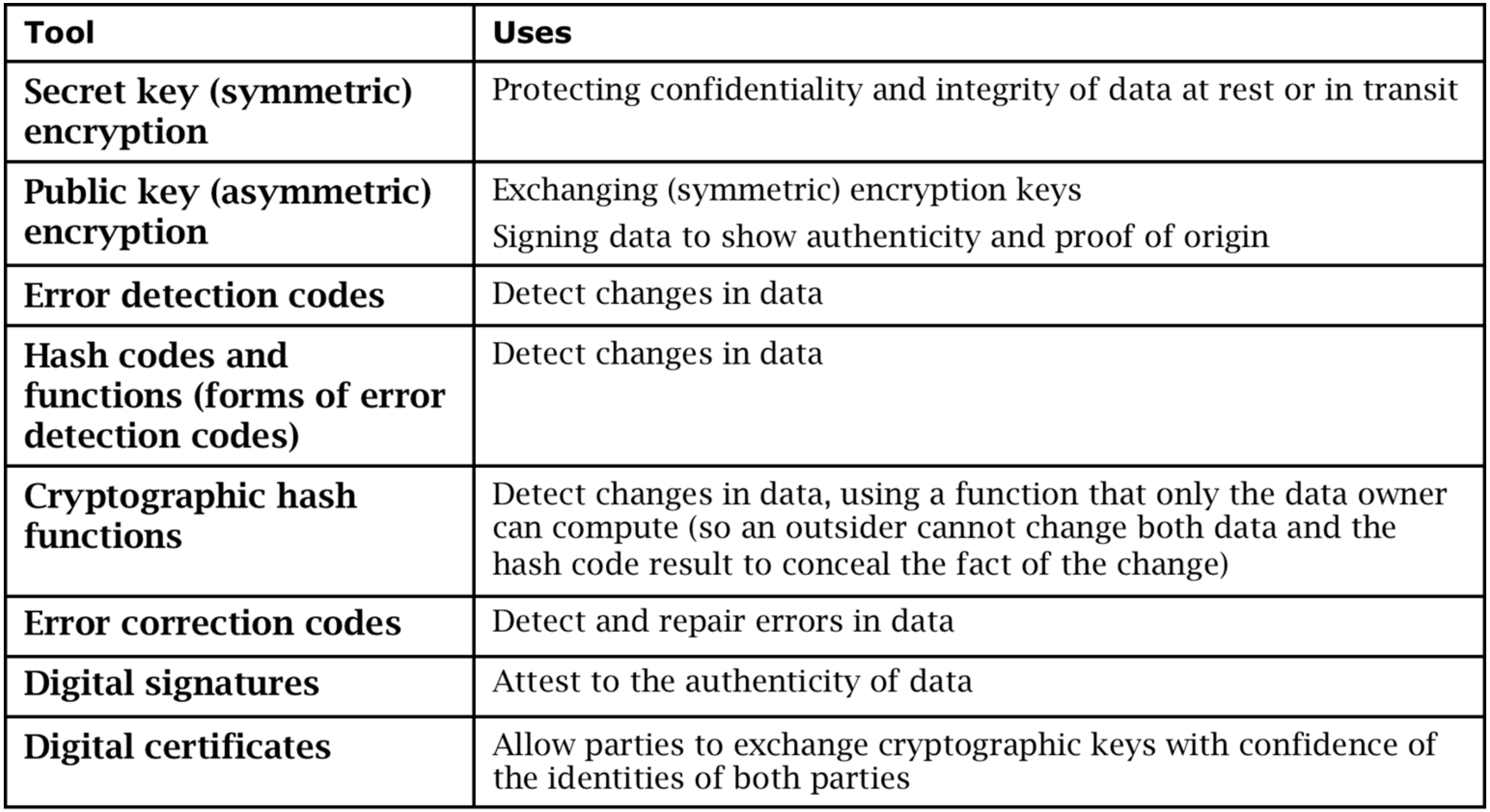
Name: Delwyn Position: Dept Manager Public key: 3AB3882C ...	hash value 48CFA
Name: Diana Position: Division Manager Public key: 17EF83CA ...	hash value 128C4

Which is Delwyn's certificate.

Complete Digital Signature

- Usage of a Digital Signature:
  - A file – encrypted
  - A secure hash code of the file is used to compute a message digest and included to show that the file has not changed
  - The recipient can recompute the hash function and compare with the message digest.
  - If the hash codes match, the recipient can conclude that the received file is the one that was signed.
  - The recipient has to know who the signer was so that the correct public key is used to decrypt the message.

Cryptographic Tool Summary



Tool	Uses
Secret key (symmetric) encryption	Protecting confidentiality and integrity of data at rest or in transit
Public key (asymmetric) encryption	Exchanging (symmetric) encryption keys Signing data to show authenticity and proof of origin
Error detection codes	Detect changes in data
Hash codes and functions (forms of error detection codes)	Detect changes in data
Cryptographic hash functions	Detect changes in data, using a function that only the data owner can compute (so an outsider cannot change both data and the hash code result to conceal the fact of the change)
Error correction codes	Detect and repair errors in data
Digital signatures	Attest to the authenticity of data
Digital certificates	Allow parties to exchange cryptographic keys with confidence of the identities of both parties

Summary

- Encryption helps prevent attackers from revealing, modifying, or fabricating messages
- Symmetric and asymmetric encryption have complementary strengths and weaknesses
- Certificates bind identities to digital signatures

Questions?

Programs and Programming

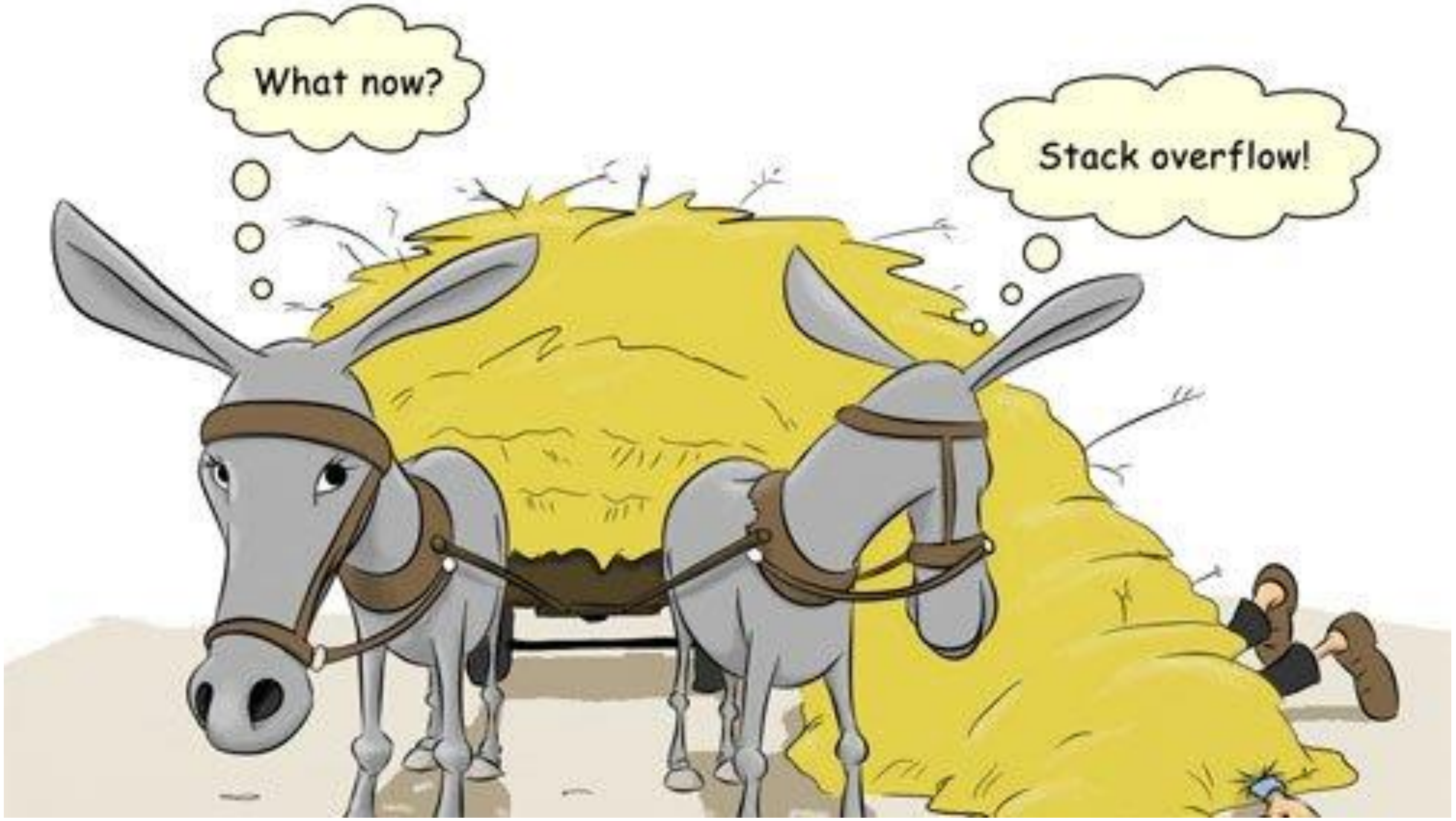
Objectives for today

- Learn about memory organization, buffer overflows, and relevant countermeasures
- Common programming bugs, such as off-by-one errors, race conditions, and incomplete mediation
- Survey of past malware and malware capabilities
- Virus detection
- Tips for programmers on writing code for security

Software Vulnerabilities

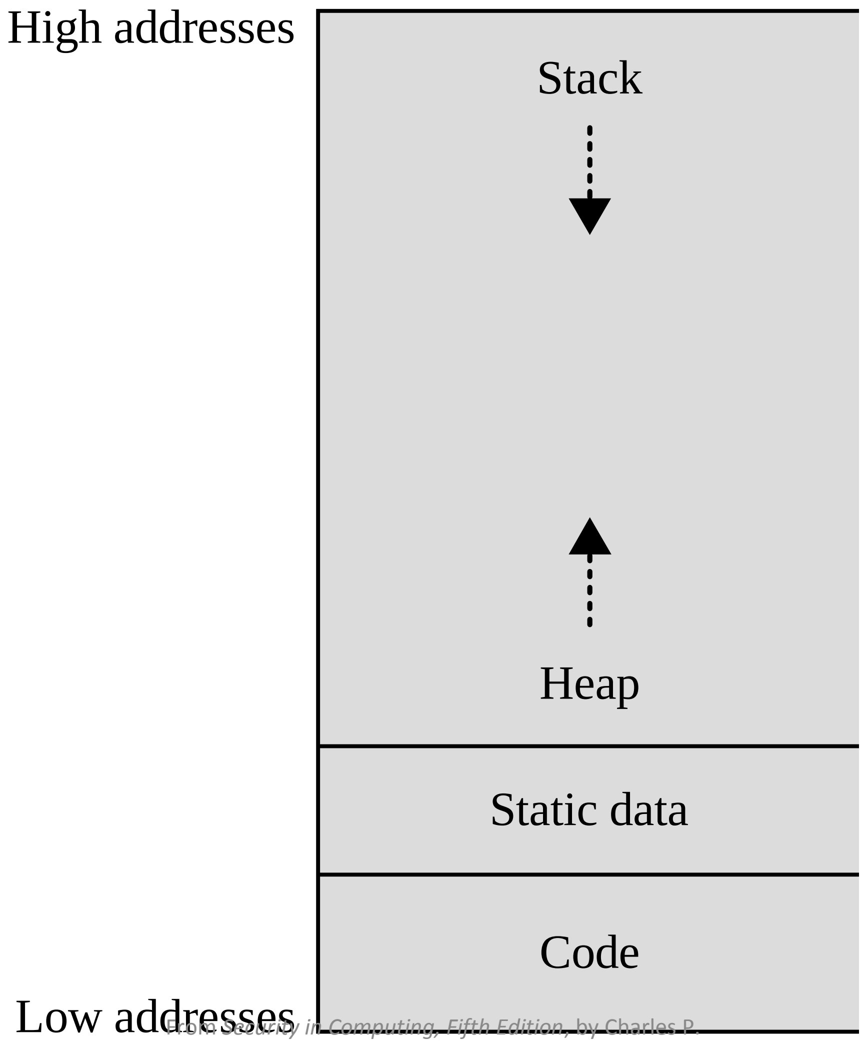
- A weakness in the program
  - May be explored by an attacker
    - Causing system to behave differently then expected
- Attacks may be based on the operating system and programming language
- Goals include manipulating the computer’s memory or control program’s execution

Buffer Overflow



https://www.linkedin.com/pulse/buffer-overflow-exploits-protection-mechanisms-roman-postanciuc/

Memory Allocation



Buffer Overflows

- Occur when data is written beyond the space allocated for it, such as a 10th byte in a 9-byte array
- In a typical exploitable buffer overflow, an attacker's inputs are expected to go into regions of memory allocated for data, but those inputs are instead allowed to overwrite memory holding executable code
- The trick for an attacker is finding buffer overflow opportunities that lead to overwritten memory being executed, and finding the right code to input

How Buffer Overflows Happen

```
char sample[10];

int i;

for (i=0; i<=9; i++)
   sample[i] = 'A';

sample[10] = 'B';
```

Programming Errors

- **Buffer overflow** attack can cause crash
  - Input is longer than variable buffer, overwrites other data
  - Example: program in C:

```
char A[8] = "";
unsigned short B = 1979;
```

Buffer Overflow (example)

Initially, A contains nothing but zero bytes, and B contains the number 1979

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

Buffer Overflow (example)

- The program attempts to store the null-terminated string "excessive“

```
strcpy(A, "excessive");
```

- "excessive" is 9 characters long and encodes to 10 bytes including the null terminator
  - but A can take only 8 bytes
  - By failing to check the length of the string, it also overwrites the value of B:

Buffer Overflow (example)

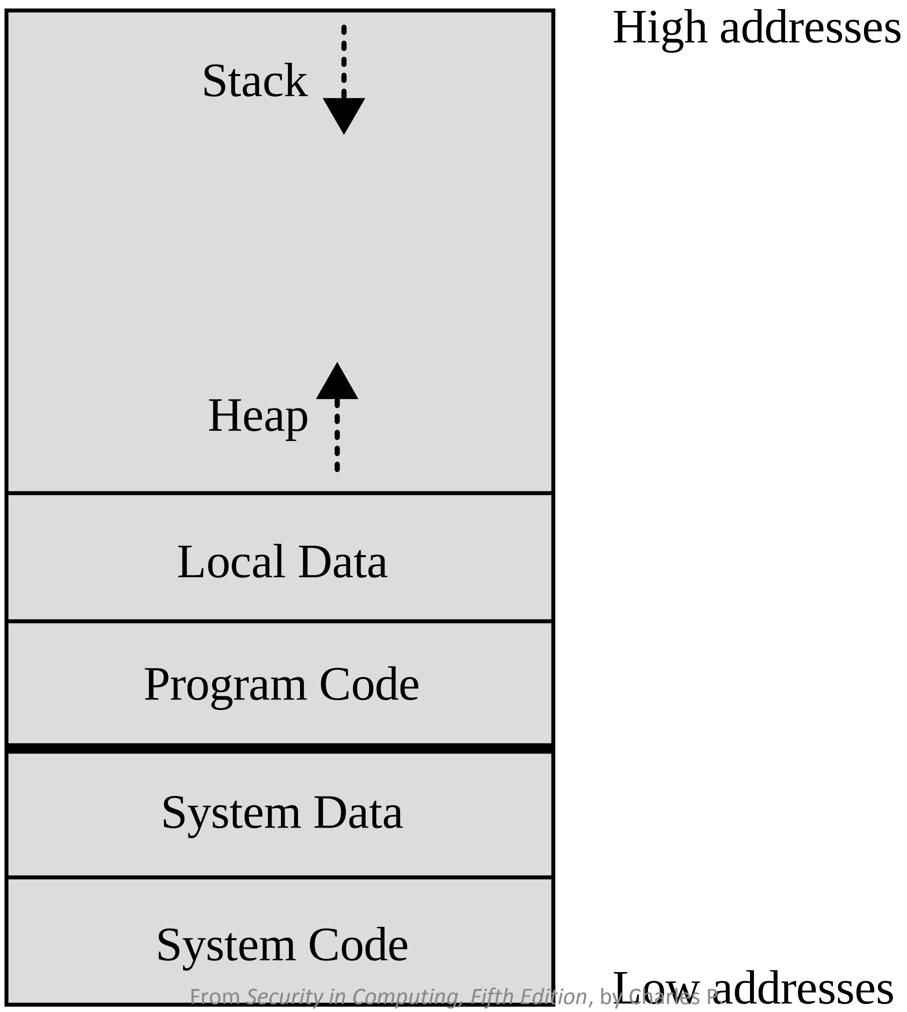
variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

Buffer Overflow (cont.)

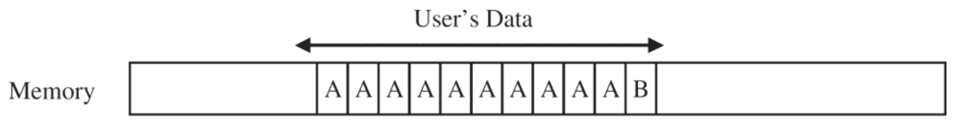
- How to prevent this?
- Replace call to strcpy with strncpy
  - strncpy takes the maximum capacity of A as an additional parameter
    - ensures that no more than this amount of data is written to A:

```
strncpy(A, "excessive", sizeof(A));
```

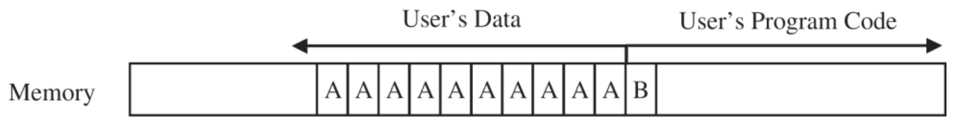
Memory Organization



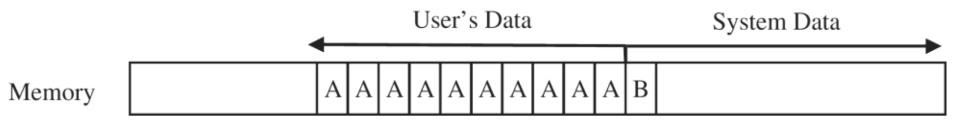
Where a Buffer Can Overflow



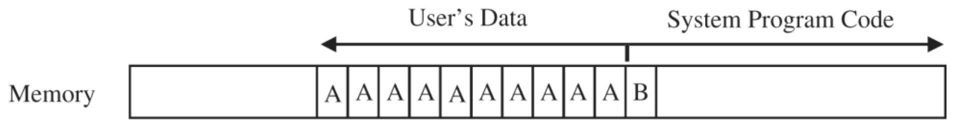
(a) Affects user's data



(b) Affects user's code



(c) Affects system data



(d) Affects system code

Buffer Overflow Attack

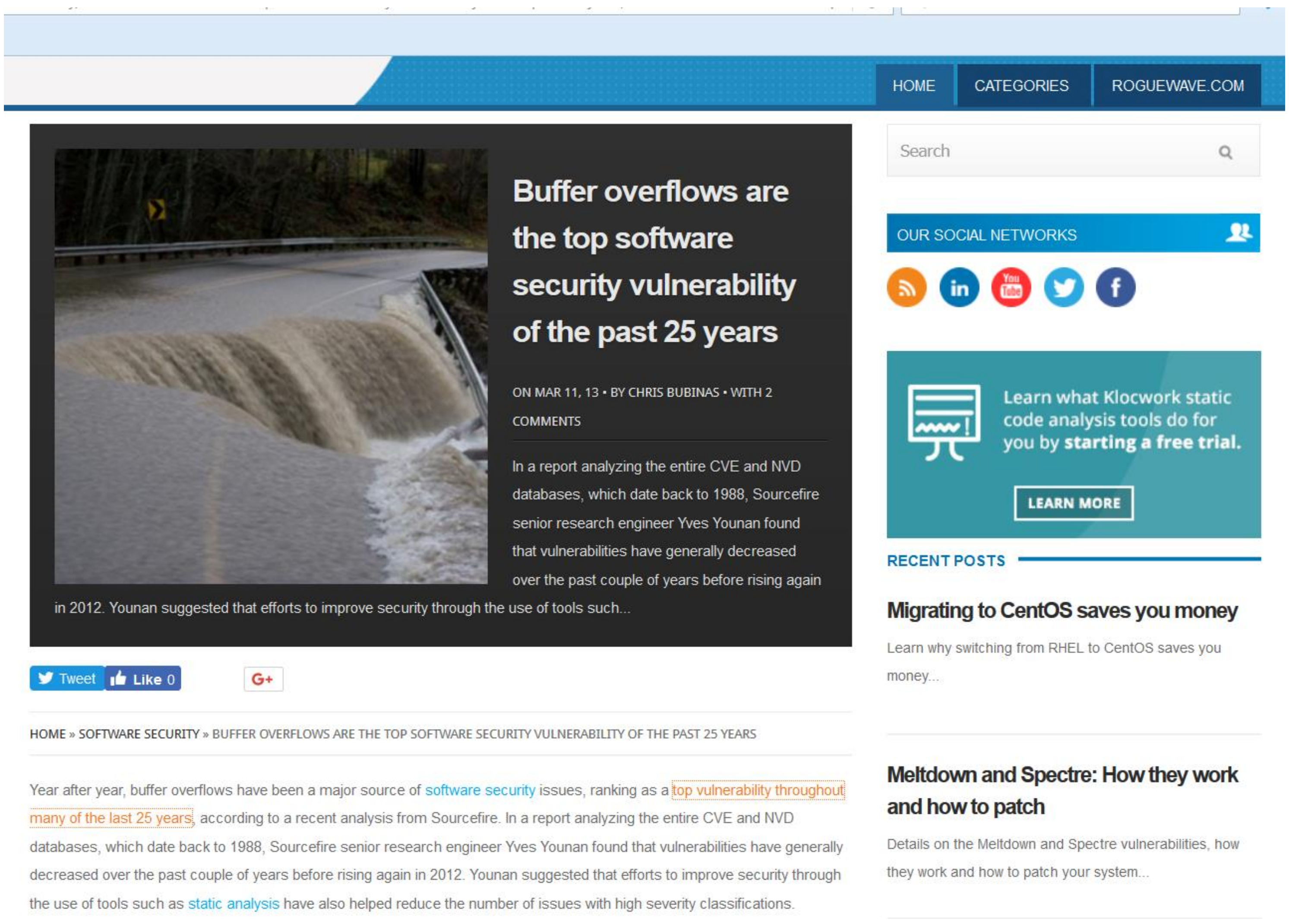
- Buffer overflow can happen by accident
  - Or a malicious attack
- Example of an attack on a web server:
  - The server accepts user input from a name field on a web page
    - Into an unchecked buffer variable
    - The attacker supplies a malicious code as input
  - The code read by the server overflows part of the application code
  - The web server now runs the malicious code

Harm from Buffer Overflows

- Overwrite:
  - Another piece of your program’s data
  - An instruction in your program
  - Data or code belonging to another program
  - Data or code belonging to the operating system
- Overwriting a program’s instructions gives attackers that program’s execution privileges
- Overwriting operating system instructions gives attackers the operating system’s execution privileges

Buffer Overflow Attack

- Buffer overflows are a major source of security issues
- Software tools help decrease server issues
  - But vulnerability still exists and exploited
- Buffer overflows account for 14% of all vulnerabilities in the last 25 years
  - But 23% of the high severity vulnerabilities and 35% of critical vulnerabilities!



Buffer overflows are the top software security vulnerability of the past 25 years

ON MAR 11, 13 • BY CHRIS BUBINAS • WITH 2 COMMENTS

In a report analyzing the entire CVE and NVD databases, which date back to 1988, Sourcefire senior research engineer Yves Younan found that vulnerabilities have generally decreased over the past couple of years before rising again in 2012. Younan suggested that efforts to improve security through the use of tools such...

HOME » SOFTWARE SECURITY » BUFFER OVERFLOWS ARE THE TOP SOFTWARE SECURITY VULNERABILITY OF THE PAST 25 YEARS

Year after year, buffer overflows have been a major source of software security issues, ranking as a top vulnerability throughout many of the last 25 years, according to a recent analysis from Sourcefire. In a report analyzing the entire CVE and NVD databases, which date back to 1988, Sourcefire senior research engineer Yves Younan found that vulnerabilities have generally decreased over the past couple of years before rising again in 2012. Younan suggested that efforts to improve security through the use of tools such as static analysis have also helped reduce the number of issues with high severity classifications.

OUR SOCIAL NETWORKS

Learn what Klocwork static code analysis tools do for you by **starting a free trial.**

LEARN MORE

RECENT POSTS

Migrating to CentOS saves you money

Learn why switching from RHEL to CentOS saves you money...

Meltdown and Spectre: How they work and how to patch

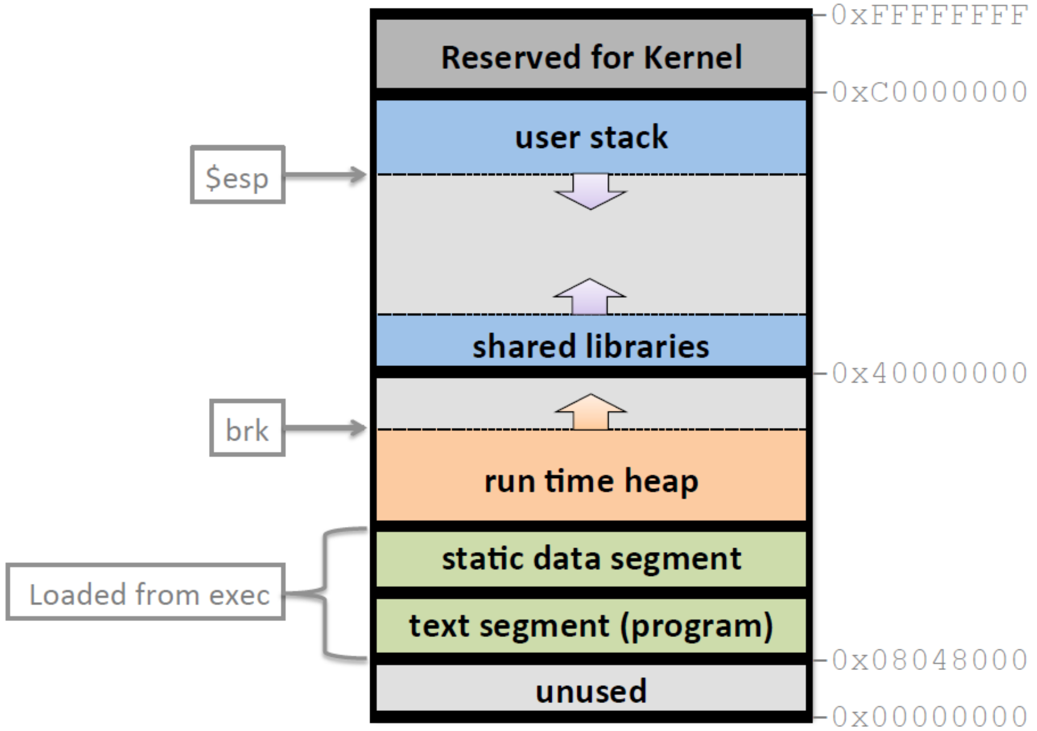
Details on the Meltdown and Spectre vulnerabilities, how they work and how to patch your system...

Code Injection

- Exploitation of a computer bug that is caused by processing invalid data.
- Code injected into a vulnerable computer program and changes the course of execution
- Results can be disastrous
  - May allow computer worms to propagate

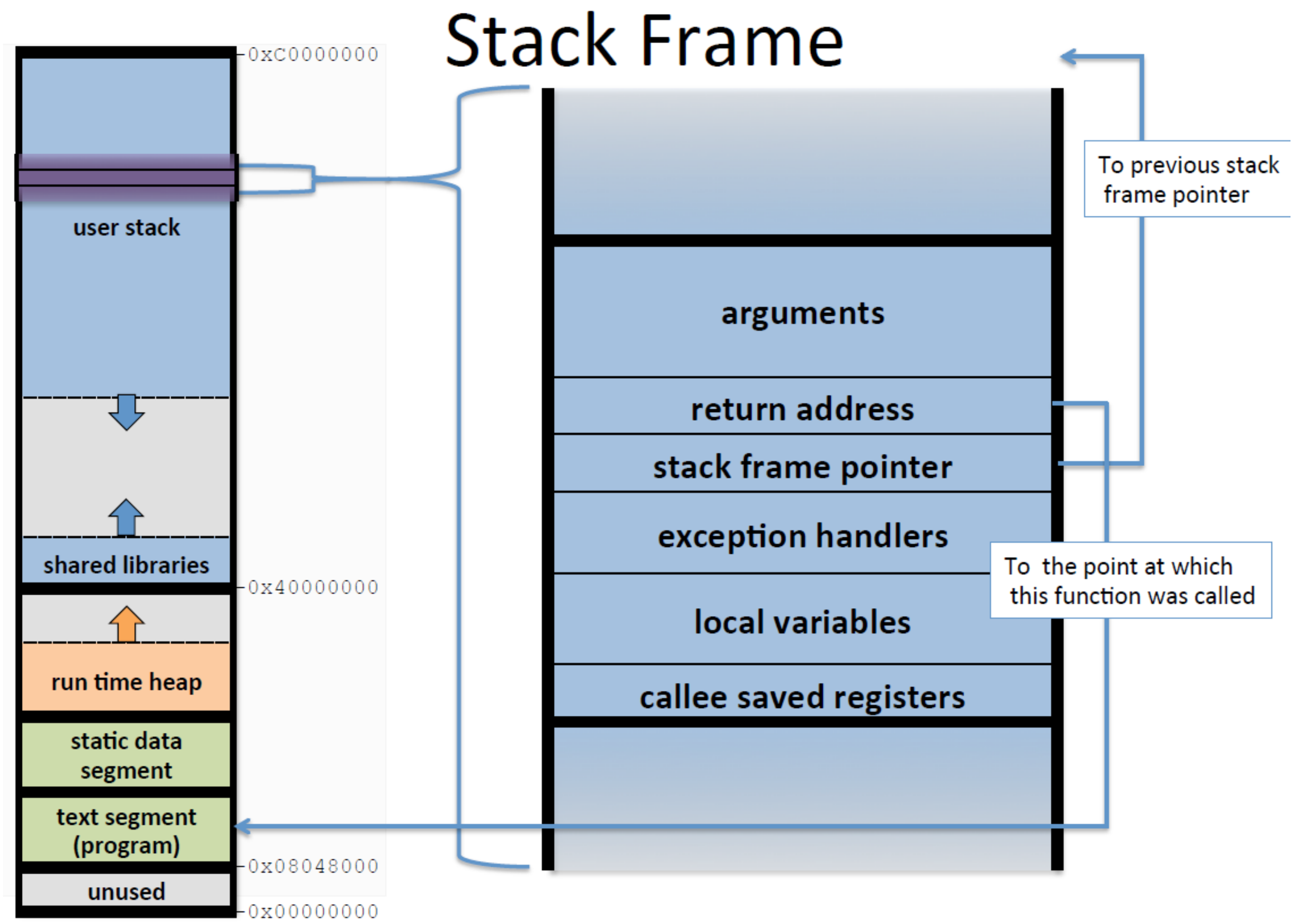
Code Injection Attack Example

Linux (32-bit) process memory layout

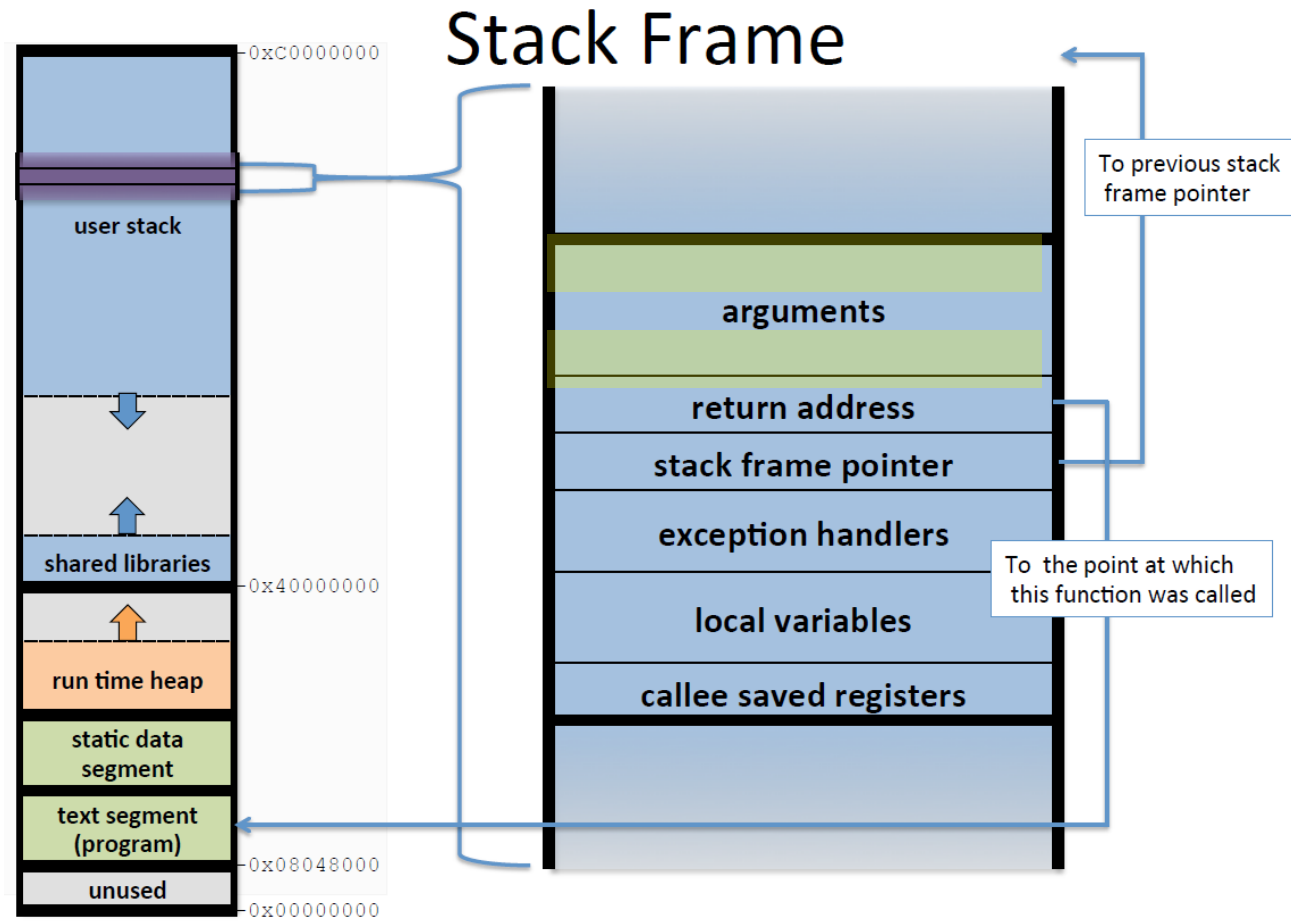


Code Injection Attack Example

- What makes frame vulnerable to attacks?



Code Injection Attack Example



Code Injection

- Basic Stack Exploit:
  - Overwriting the return address allows an attacker to redirect the flow of program control.
  - Instead of crashing, this can allow *arbitrary* code to be executed.

Code Injection

- Basic Stack Exploit example:
  - attacker chooses malicious code he wants executed (“shellcode”), compiles to bytes
  - includes this in the input to the program so it will get stored in memory somewhere
  - overwrites return address to point to it.

Code Example 1

- Is this function safe?

```
void vulnerable() {
    char buf[64];
    …
    gets(buf);
    …
    }
```

- How can you make it safer?

Code Example 1 (cont.)

- Is this function safe?

```
void safe() {
        char buf[64];
        …
        fgets(buf, 64, stdin);
        …
        }
```

- Can we make it safer?

Code Example 1 (cont.)

- Is this function safe?

```
void safe() {
        char buf[64];
        …
        fgets(buf, 64, stdin);
        …
        }
```

- What happens when the function changes over time?

Code Example 1 (cont.)

- Function grows after a while...

```
void safe() {
    char buf[64];
    …
    …
    …
    …
    …
    fgets(buf, 64, stdin);
    …
    }
```

Code Example 1 (cont.)

- A bigger buffer may be needed, a change may (eventually) occur:

```
void safe() {
        Char buf[64];
        …
        …


        …
        …
        fgets(buf, 128,stdin)
```

- How can we make it safer?

Code Example 1 (cont.)

```
void safer() {
        char buf[64];
        …
        fgets(buf, sizeof buf, stdin);
        …
        }
```

Code Example 2

- Is this function safe?

```
void vulnerable(int len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

```
memcpy(void *s1, const void *s2, size_t n);
```

Code Example 2

- Size_t is an unsigned integer type
- All signed integer negative values change to a high positive number when converted to an unsigned type
  - In libc routines, such as memcpy
- How can an attacker exploit this?

Code Example 2

- Size_t is an unsigned integer type
- All signed integer negative values change to a high positive number when converted to an unsigned type
- Malicious users can often specify negative integers through various program interfaces
  - undermine an application's logic
- This happens commonly when a maximum length check is performed on a user-supplied integer, but no check is made to see whether the integer is negative

Code Example 2

```
void safe(size_t len, char *data) {
  char buf[64];
  if (len > 64)
    return;
  memcpy(buf, data, len);
}
```

Code Example 3

-

```
void f(size_t len, char *data) {
  char *buf = malloc(len+2);
  if (buf == NULL) return;
  memcpy(buf, data, len);
  buf[len] = '\n';
  buf[len+1] = '\0';
}
```

- Is it safe?
  - No, vulnerable!
  - If len = 0xffffffff, then program allocates only 1 byte
    - Overflows

Causes of Software Vulnerabilities

- **Human-errors**: human error is a significant source of software vulnerabilities and security vulnerabilities
  - Solution? Use automated tools
- **Awareness**: programmers not be focused on security
  - May be unaware of consequences
  - Solution: Learn about common types of security flaws
- **Design flaws:** software and hardware have design flaws and bugs
  - May not be designed well for security
    - Use better languages (Java, Python…)

Causes of Software Vulnerabilities

- **Complexity**: software vulnerabilities rise proportionally with complexity
  - Solution: design simple and well-documented software
- **User input**: accepting user input by internet can introduce software vulnerabilities
  - Data may be incorrect or fraudulent
    - Data can be designed to attack the receiving system
  - Solution: sanitize user input

Testing for Software Security Issues

- What makes testing a program for security problems difficult?
  - If programmer doesn’t make mistake, program will run correctly
  - We need to test for the *absence* of something
    - Security is a negative property!
      - “nothing bad happens, even in really unusual circumstances”
  - Normal inputs rarely stress security-vulnerable code

Testing for Software Security Issues

- How can we test more thoroughly?
  - Use random inputs
  - Create a testing plan
    - Allows defining a range of inputs

https://www.avyaancom/blog/why-hire-a-software-security-testing-company/

http://www.testnbug.com/2015/02/software-testing-types/

Testing for Software Security Issues

- How can we test more thoroughly?
  - Use random inputs
  - Create a testing plan
    - Allows defining a range of inputs
- How do we tell if a problem was found?
  - Crash or other deviant behavior;
    - now more expensive checks justified

https://www.avyaan.com/blog/why-hire-a-software-security-testing-company/

http://www.testnbug.com/2015/02/software-testing-types/

Questions?