

# Lecture 19: Security II

Professor G. Sandoval

Some slides derived from : Tanenbaum/Bo, and Brendan Dolan-Gavitt  
Thanks !!

# Security

- ➔ • Basics of Cryptography
  - Authentication
  - Software Security

# Kerckhoffs's Principle

- Basic principle: it should not matter if the cryptographic *algorithm* is known by everyone
- The only secret information should be a *secret key* chosen by the participants
  - Less information to keep secret
  - If the key is revealed, we can just change it
- Restated by Claude Shannon: "The enemy knows the system"

# Secret Key Cryptography

- Two parties who want to communicate agree on a *secret key* shared between them
- Then they use *encryption* and *decryption* functions:
  - $E(\text{Data}, \text{Key}) = \text{Encrypted Data}$
  - $D(\text{Data}, \text{Key}) = \text{Decrypted Data}$
  - $D(E(\text{Data}, \text{Key}), \text{Key}) = \text{Data}$

# Example: Caesar Cipher

- Algorithm: take each letter and shift it forward in the alphabet by n letters
- Secret key: the number to shift by (1-25)
- For example, ROT13: shift by 13 letters
  - ATTACK AT DAWN
  - NGGNPX NG QNJA

# Secret Key Cryptography

- Modern secret key crypto is much more sophisticated
- Algorithms like AES (the Advanced Encryption Standard) employ substitutions and permutations so that the output has no relationship to input unless the secret is known
  - As far as we know! We have been wrong before...
- Keys are generally 256 **random** bits – much too large to try all combinations

# Public Key Cryptography

- Secret key cryptography was the only kind that existed until the 1970s
- It's inconvenient! Two parties have to somehow securely transmit (or agree on) a secret key
- Particularly on the Internet, this is impractical
  - Imagine having to visit an Amazon office to get a secret key in order to shop online...

# Public Key Cryptography

- Instead of a single secret key, we now have a key with two parts: a public key and a private key
  - $E(\text{PubKey}, \text{Data}) = \text{Encrypted Data}$
  - $D(\text{PrivKey}, \text{Data}) = \text{Decrypted Data or Data}$
  - **$D(\text{PrivKey}, E(\text{PubKey}, \text{Data})) = \text{Data}$**
- So now to communicate securely, we just need to know someone's private key



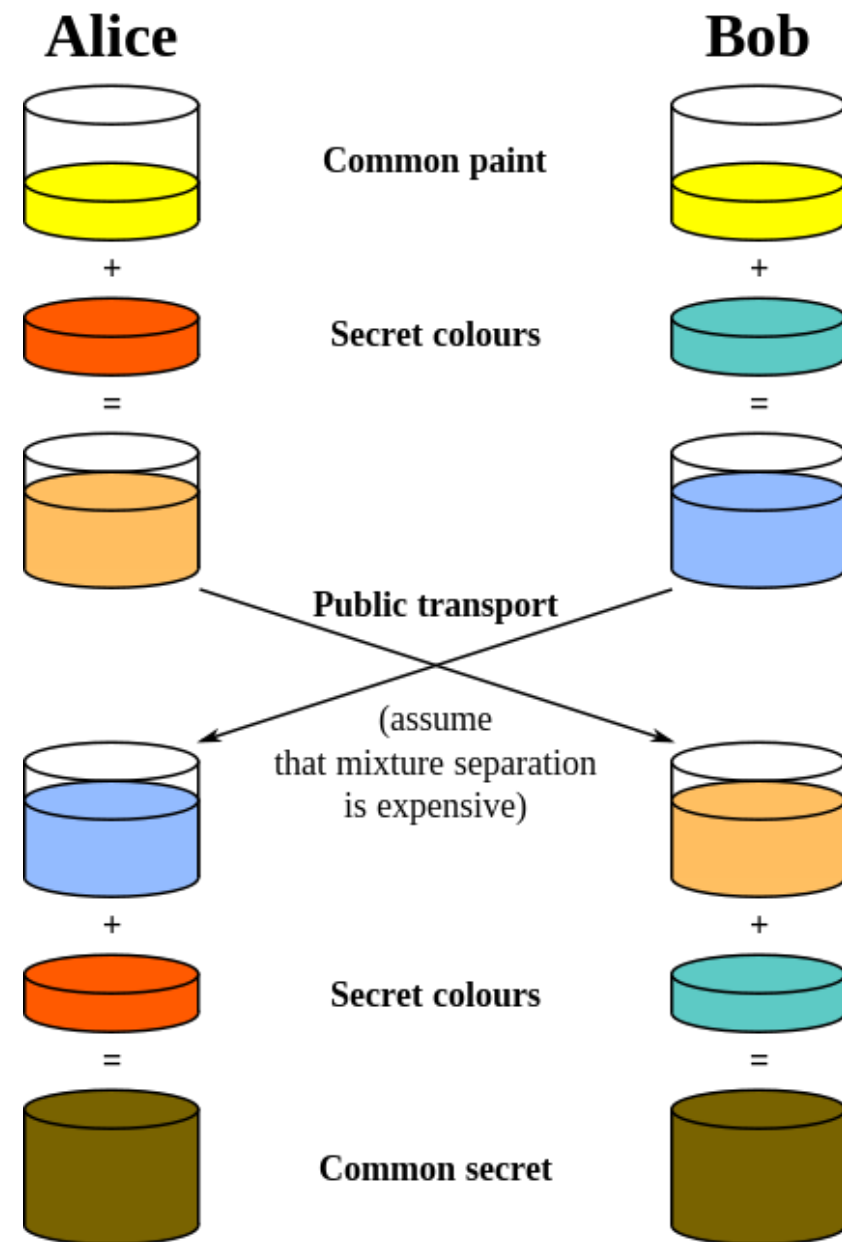
# Public Key Crypto Algorithms

- Some examples:
  - RSA (1977)
  - Elliptic Curve Algorithms

# Key Exchange

- A slightly different twist on this: can we use only public communications to allow two people to agree on a secret key?
- Surprisingly, the answer is yes!
- Whitfield Diffie and Martin Hellman first showed this was possible in 1976

# Diffie-Hellman by Analogy



# Hash Functions

- We saw before that one-way functions can let us do useful things like store capabilities
- Basic idea:  $\text{Hash}(x) = y$
- **Preimage resistance:** It should be very difficult to take  $y$  and figure out  $x$
- **Second preimage resistance:** It should be very difficult to find another value  $z$  where  $\text{Hash}(z) = y$
- **Collision resistance:** It should be very difficult to find  $x_1$  and  $x_2$  such that  $\text{Hash}(x_1) = \text{Hash}(x_2)$
- Examples: MD5, SHA-1, SHA-3

# Security

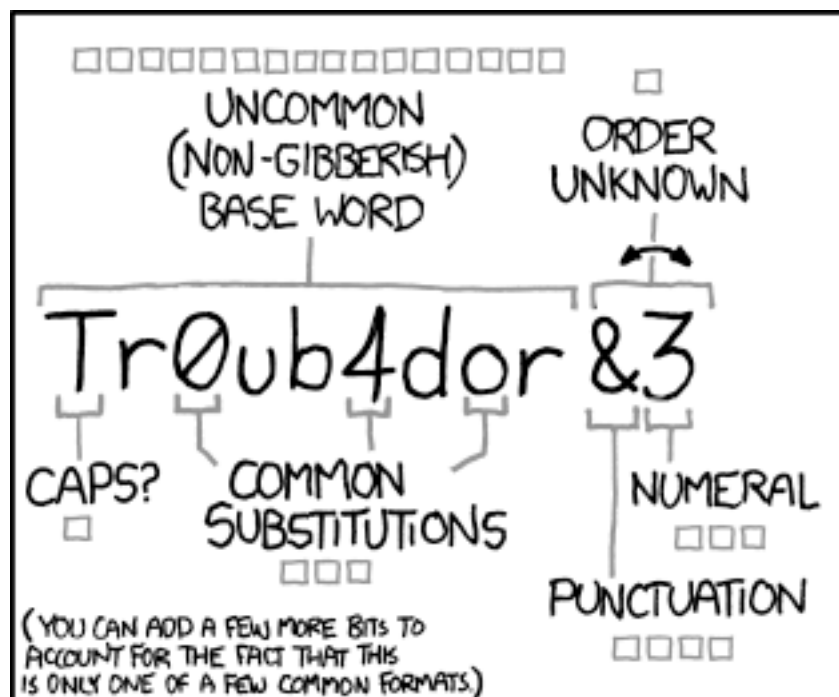
- Basics of Cryptography
- • Authentication
- Software Security

# Authentication

- To prove you are who you say you are, we usually use one of three things:
  - Something you know
  - Something you have
  - Something you are

# Passwords

- "Something you know" – one of the simplest and oldest forms of authentication, and still ubiquitous
- Generally easy to change
- Very easy to implement, since you can just ask for a short string and compare it with the one you have stored



~28 BITS OF ENTROPY

□□□□□□□□  
□□□□□□□□ □  
□□□ □□□  
□□□□ □


$2^{28} = 3 \text{ DAYS AT } 1000 \text{ GUESSES/SEC}$

(PLAUSIBLE ATTACK ON A WEAK REMOTE  
WEB SERVICE. YES, CRACKING A STOLEN  
HASH IS FASTER, BUT IT'S NOT WHAT THE  
AVERAGE USER SHOULD WORRY ABOUT.)

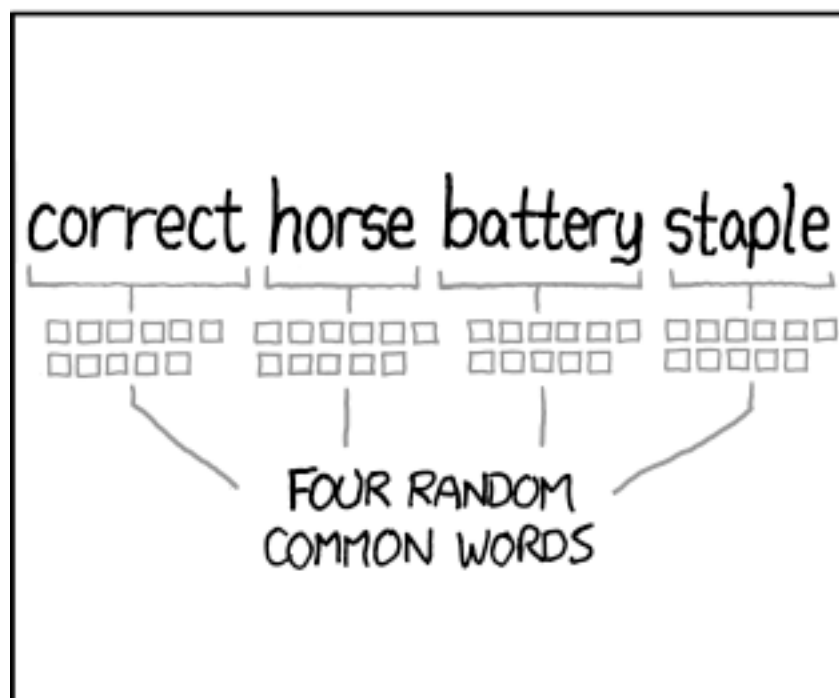
DIFFICULTY TO GUESS:  
**EASY**

WAS IT TROMBONE? NO,  
TROUBADOR. AND ONE OF  
THE 0s WAS A ZERO?

AND THERE WAS  
SOME SYMBOL...



DIFFICULTY TO REMEMBER:  
**HARD**



~44 BITS OF ENTROPY

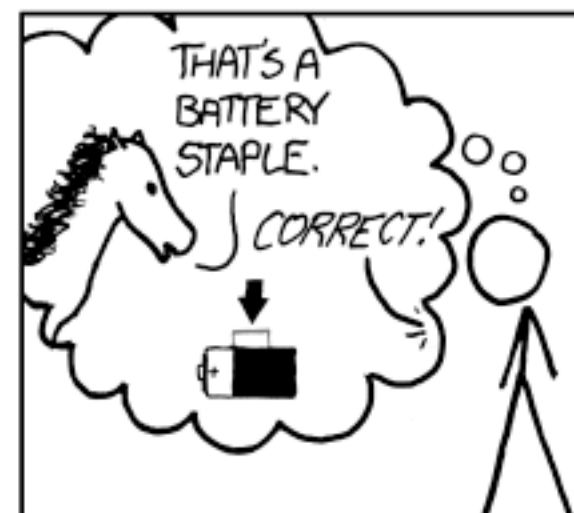
□□□□□□□□□□  
□□□□□□□□□□  
□□□□□□□□□□  
□□□□□□□□□□

$2^{44} = 550 \text{ YEARS AT } 1000 \text{ GUESSES/SEC}$

DIFFICULTY TO GUESS:  
**HARD**

THAT'S A  
BATTERY  
STAPLE.

CORRECT!



DIFFICULTY TO REMEMBER:  
YOU'VE ALREADY  
MEMORIZED IT

THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED  
EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS  
TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.



# Password Storage

- It's generally a bad idea to store passwords directly
- If someone gets ahold of the password file, they can read all passwords
- This is particularly bad since people tend to **re-use** their passwords in multiple places...

# Password Storage

- Instead, we use a cryptographic hash function on the password
- We compute (for example):
  - $\text{SHA1}(\text{"goodpassword"}) =$   
7e5ce399fbe3713ec7f6aae370448cdf990f0aaa
- Then we only store  
7e5ce399fbe3713ec7f6aae370448cdf990f0aaa
- Now to check the password someone entered, we hash it and compare the hashes instead

# How Passwords are Broken

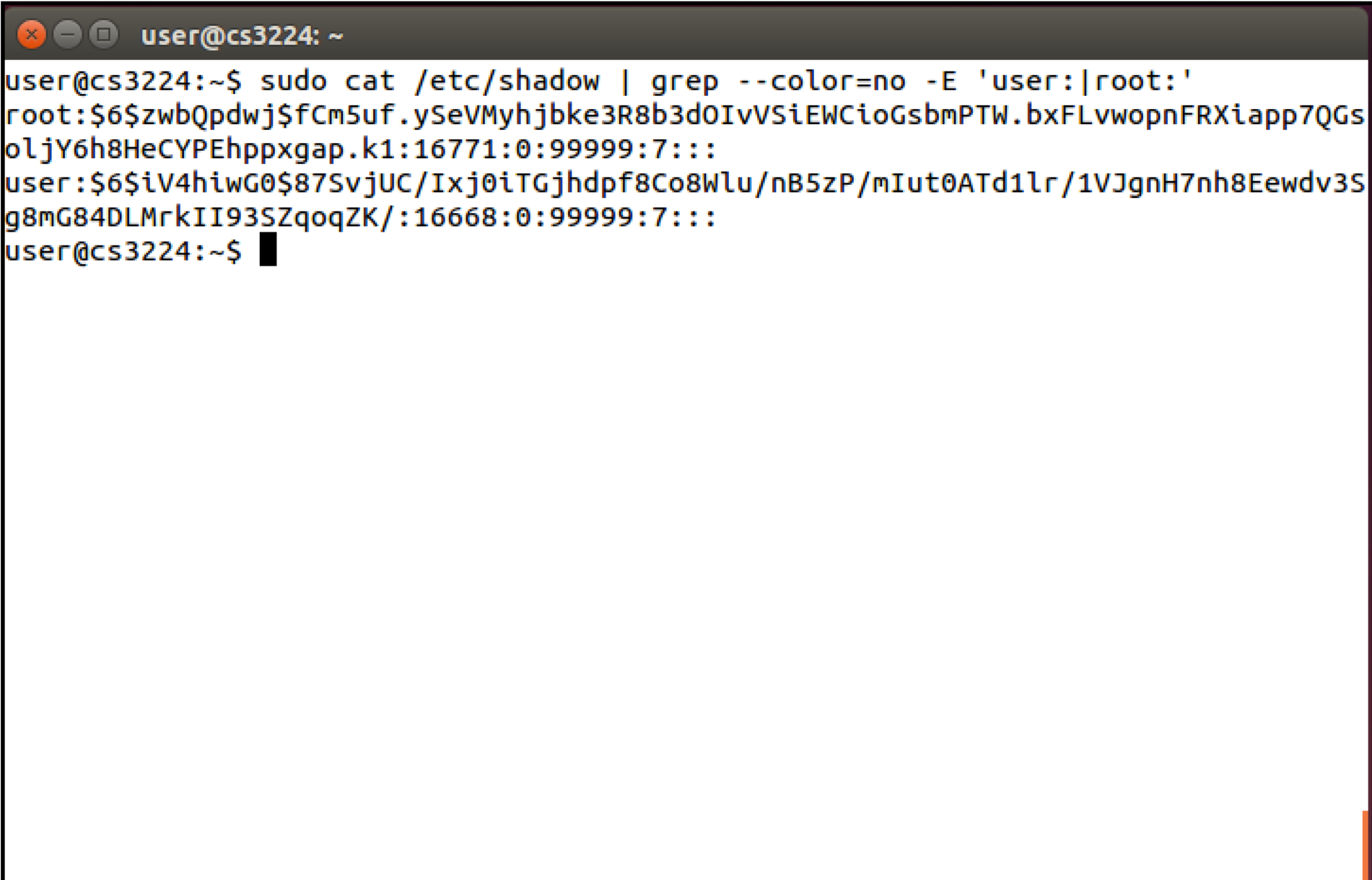
- Assuming they only have a file with password hashes, how can an attacker find out the original passwords?
- Password hashes nowadays are broken primarily in two ways:
  - **Precomputation:** Spend a lot of time beforehand computing the hash **of every possible password**, then just look up each hash in your table
  - **Dictionary Attacks:** Guess the **most common passwords** (e.g. using a dictionary), hash them, and compare

# Password Salts

- To avoid precomputation attacks, we can use a *salt*
- Instead of computing and storing Hash(Password), we instead:
  - Generate a random string (a **salt**)
  - Compute Hash(Salt+Password)
  - Store on disk Salt, Hash(Salt+Password)
- Now to do a precomputation, you have to store not just the hash of every possible password (hard but doable) but the hash of every possible password+salt (way too much space)

# UNIX Password Files

- UNIX password files are hashed and stored with salts:



```
user@cs3224: ~  
user@cs3224:~$ sudo cat /etc/shadow | grep --color=no -E 'user:|root:'  
root:$6$zwbQpdwj$fCm5uf.ySeVMyhjbke3R8b3d0IvVSiEWCioGsbmPTW.bxFLvwopnFRXiapp7QGso  
ljY6h8HeCYPEhppxgap.k1:16771:0:99999:7:::  
user:$6$iV4hiwG0$87SvjUC/Ixj0iTGjhdpf8Co8Wlu/nB5zP/mIut0ATd1lr/1VJgnH7nh8Eewdv3S  
g8mG84DLMrkII93SZqoqZK/:16668:0:99999:7:::  
user@cs3224:~$
```

# Time- and Memory-Hard Password Hashing

- Cryptographic hash functions were developed to be *fast*
- This is good for many uses of them, but bad for passwords: it lets an attacker try **billions** of passwords per second
- To avoid this, people have developed hash algorithms that are deliberately very slow (and cannot be sped up)
  - One example is **bcrypt**
- We may also want it to take a lot of memory
  - A newer algorithm called **scrypt** does this

# One-Time Passwords

- An alternative to having a single password is to use a different password every time
- Even if an attacker eavesdrops on a connection and learns the password, it doesn't do them any good
- A password list can be exchanged securely (e.g., banks sometimes send them in the mail)



# Nordea Bank one-time password list

Card no. 9008759225

01-IU 4455	11-LN 3207	21-GR 2807	31-WD 7558
02-OH 7438	12-UF 6838	22-RX 1323	32-WK 7765
03-NU 2365	13-SL 7027	23-PJ 7191	33-KY 0452
04-II 8859	14-RN 7894	24-WZ 6752	34-MF 0965
05-IQ 0388	15-BE 1806	25-XQ 1597	35-CN 4260
06-WQ 3572	16-ZL 1769	26-IM 1498	36-TZ 5047
07-SJ 7844	17-QM 3891	27-MI 0762	37-SM 7916
08-IV 6424	18-TP 9892	28-TM 0987	38-KQ 6426
09-GK 9623	19-US 1854	29-PD 5288	39-ES 5992
10-WU 5578	20-TH 5502	30-UH 5939	40-VJ 3515

41-GI 7146	51-QH 8027	61-ZT 0271	71-FD 4730
42-FP 3297	52-CJ 6502	62-KG 2048	72-VV 9845
43-VB 9276	53-JC 5925	63-SI 0550	73-GM 7628
44-VY 6679	54-MM 3286	64-UT 4694	74-YZ 1691
45-SW 5187	55-RT 2311	65-UU 2301	75-CG 5697
46-AV 6100	56-BT 1704	66-RW 8490	76-SF 2107
47-GJ 3536	57-JO 0729	67-ZP 3758	77-DT 7470
48-OC 2813	58-YM 6206	68-DR 8424	78-JU 0047
49-ZM 9175	59-GH 3024	69-BL 8702	79-ON 3292
50-NT 9622	60-JQ 3697	70-LP 7668	80-YN 9361

81-LA 6865	91-NR 5144	101-WJ 7731	111-PV 1825
82-DK 1261	92-WY 0917	102-KN 6907	112-QF 4198
83-VS 1837	93-RL 2605	103-IL 8153	113-GU 7578
84-IR 2804	94-GO 0523	104-DE 0478	114-EK 8973
85-ZX 2250	95-RE 3860	105-HF 7664	115-YH 2589
86-MO 9659	96-XE 8676	106-XZ 0754	116-YD 6548
87-PR 5720	97-ZG 4926	107-UC 0587	117-TO 0547
88-LB 0288	98-TJ 9531	108-LK 9936	118-XO 0678
89-GE 5036	99-CP 3886	109-HW 7655	119-FJ 8151
90-CZ 4174	100-GS 9595	110-AJ 9792	120-OL 0586



# Lamport Hash Chain

- Instead of mailing out passwords each time, we can get many of the benefits using a hash chain
- The idea is to use a one-way function to create a sequence of hashes
- If you repeatedly apply a hash to some starting value, you get a sequence that is:
  - Easy to compute starting from the seed going forward
  - Hard to compute starting from a hash value and going backward

# Lamport Hash Chain

- Initially, user chooses a password  $p$
- Compute  $f^k(p) = f(f(f(\dots f(p))))$  and store this value on the server
- To log in, user computes and sends  $h = f^{k-1}(p)$
- Server checks that  $f(h)$  matches its stored value, and then replaces the current stored value with  $h$

# Challenge-Response

- Another way to avoid sending a password directly over the network is *challenge response* authentication
- The idea is that the server sends a *challenge* (a random value)
- Client then uses their secret and the challenge to compute a *response*
- Server can now verify that the response is the correct one for that challenge

# Challenge-Response

- One simple way to implement challenge response is to once again use a one-way function
- Client and server each know some secret  $s$  (say, a password)
- Server computes a random  $r$  and sends it to the client
- Client computes  $f(r+s)$  and sends it to the server
- Server can do the same computation to verify
- Value sent over the network is useless to an attacker since the challenge ( $r$ ) will not be reused

# Authentication

- To prove you are who you say you are, we usually use one of three things:
  - Something you know
  - Something you have
  - Something you are

# Physical Tokens

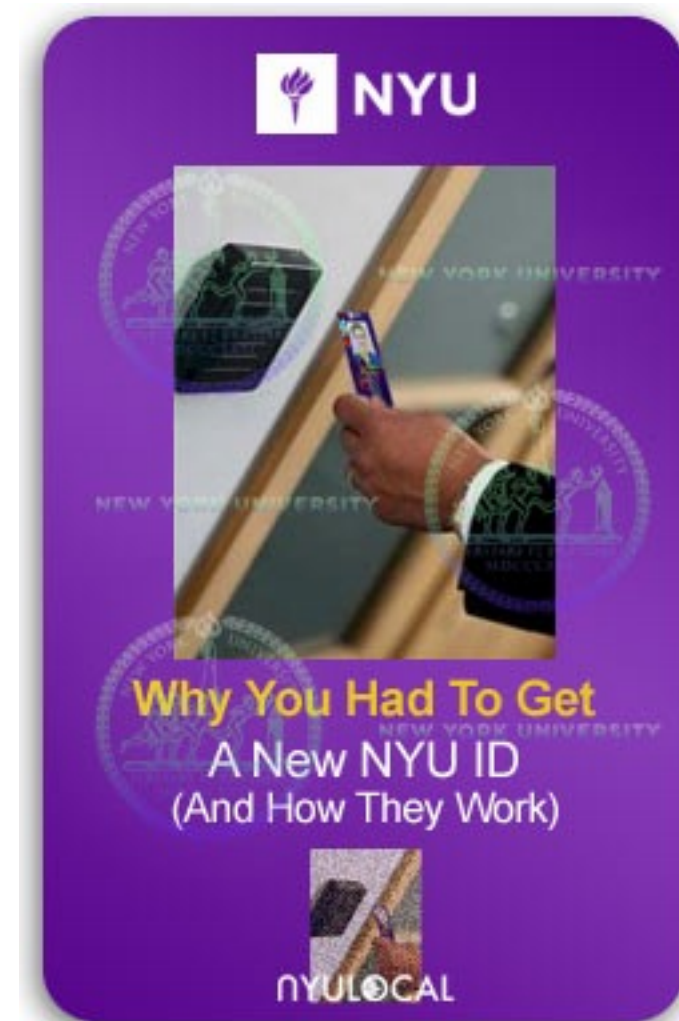
- Oldest examples: physical keys
- Basically – something you physically possess that grants you access to something
- There are lots of these now: ATM cards, smart cards, RFIDs, secure tokens, smartphones

# Smart Cards

- Credit card sized
- Contain a chip that can do some simple cryptographic computations and stores a secret key
- The reader can then ask the chip to do cryptographic operations *without* revealing the key to the reader

# RFID Authentication

- Radio communications can be used instead of direct contact
- When you put the card near a reader, the reader actively probes
- The card can collect energy from the incoming radio waves, compute a response, and send it back



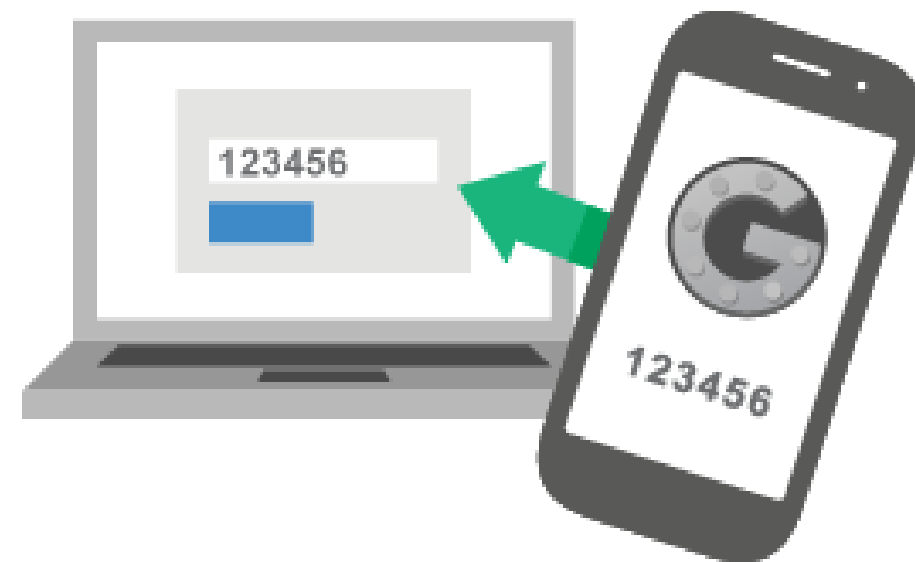
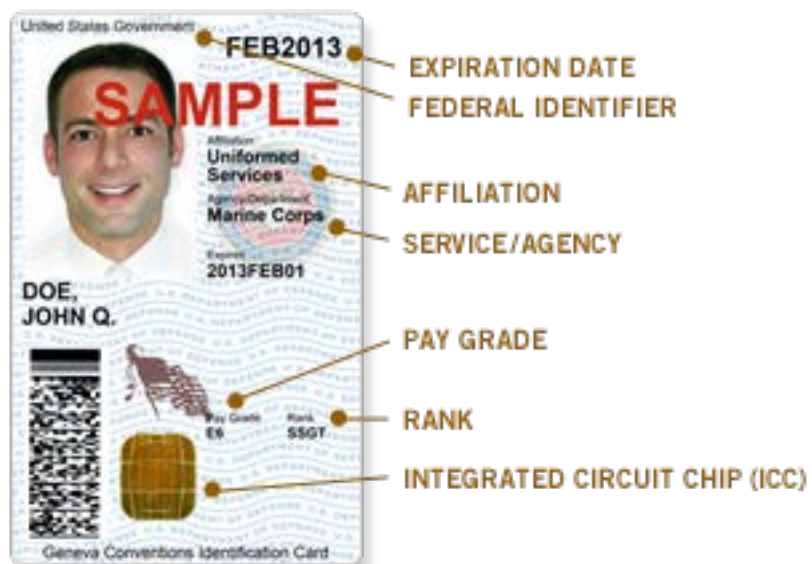


# The Great Seal Bug

- An early predecessor to RFID technology was the *great seal bug*
- This was a carved wooden seal presented by the Soviets to the US in 1945
- If radio waves at the right frequency were sent to the device, it activate the circuit and send sound back
- Entirely passive listening device – very hard to detect!



# Physical Tokens



# Authentication

- To prove you are who you say you are, we usually use one of three things:
  - Something you know
  - **Something you have**
  - Something you are

# Biometrics

- Idea: measure something intrinsic to a person that is different for everyone
- Things you can measure (partial list):
  - Keystroke timing
  - Fingerprints
  - Finger length
  - Retina pattern
  - Face recognition (iPhone Face ID)
  - Voiceprint
  - Gait
  - DNA?

# Biometrics

- Two phases: *enrollment* and *verification*
- Enrollment consists of taking the measurements of a user and converting them into digital form
  - Often measurements are taken multiple times to account for measurement error and natural variation
- At verification time, redo the measurement, and decide if it's **close enough** (will almost never be exactly the same)
- Note the tradeoff here: risk of rejecting a valid user vs accepting an invalid user



# Biometric Downsides

- Many biometric features are not very stable
  - Voiceprint may change if you get a cold
  - Gait may change if you twist your ankle
- Some features may not be very acceptable to users
  - E.g., blood sample collection

# Biometric Revocation Problem

- The biggest problem with biometrics is that they are not *revocable*
- If my password is stolen, I get a new password
- If my fingerprints are stolen, ???



**Matthew Green**

@matthew\_d\_green



Following

I woke this morning to find my 7 y/o leveraging my finger onto the TouchID sensor of my phone. Maybe time to go back to passwords.

RETWEETS

2,466

LIKES

1,834



8:30 AM - 25 Nov 2014





# Security

- Basics of Cryptography
- Authentication
- • Software Security

# Software Security

- One final aspect to OS security is *software security* – identifying and preventing programming flaws that could let an attacker take control
- Many of these are caused by the fact that languages currently in use are not *memory safe* – it is possible to write to data outside of program variables
- A big offender here is C/C++

# Stack Buffer Overflows

- We saw one of these earlier in the semester
- Recall the standard stack frame:
  - Local variables
  - Saved frame pointer (optional)
  - Return address
- If we try to store too much data in a local stack variable (e.g. a character array) we will overwrite the frame pointer and return address
- When the ret instruction is executed, it will jump to somewhere controlled by user input

# Classic Buffer Overflow

```
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    char buf[64];
    strcpy(buf,argv[1]);
    printf("Input:%s\n",buf);
    return 0;
}
```

# Operating System Defenses

- OSes can be designed to make problems in user-level applications harder to exploit
- These generally don't make attacks on software impossible, but they can make them much more difficult
- This is something of an arms race – defenders come up with new mechanisms, attackers find ways around them

# Stack Canaries / Cookies

- Idea: put a special value in between local variables and the return address so that overflowing a local buffer can be detected
- Upon entering the function, set a randomly-generated *cookie* value on the stack and store a backup copy elsewhere
- Before executing a ret, check the stack cookie value against the backup and raise an error if it fails

# Address Space Layout Randomization

- Exploiting a buffer overflow typically requires knowing about the precise layout of memory
- For example, we may need to know where the stack is located, or where a certain library has been loaded
- Thus, to make attackers' lives more difficult, we can place the program, stack, and libraries at random locations each time the program starts

# ASLR and Address Space

- We can estimate the amount of randomness provided by ASLR by counting the number of possible locations to load things
- On a 32-bit system, address space is not very large, so there are not very many ways to randomize
- In 2004, researchers showed that on 32-bit systems, ASLR only has about 16 bits of entropy (65,536 possible values)
- The correct location can be guessed in a matter of seconds by just trying each possibility

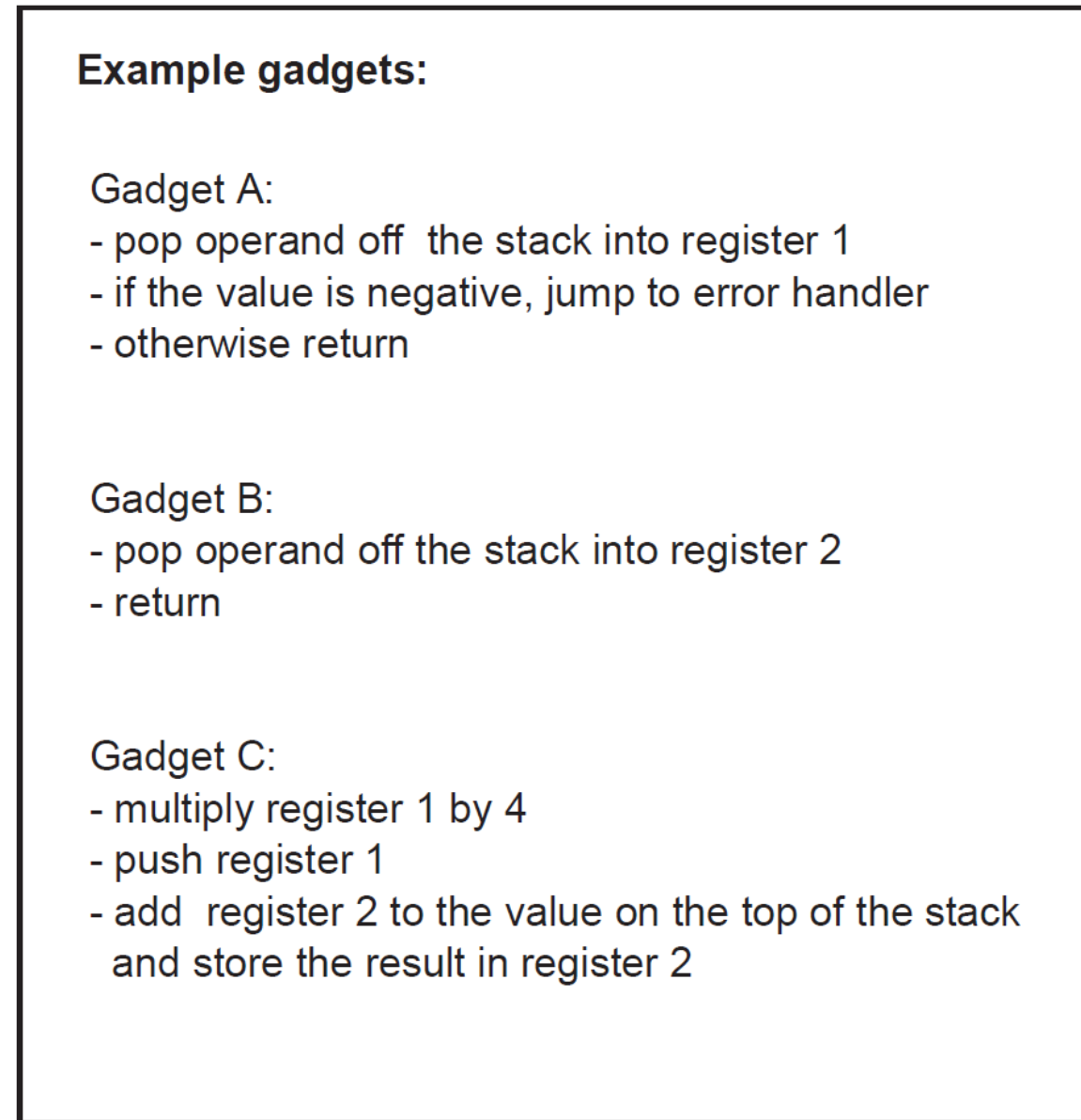
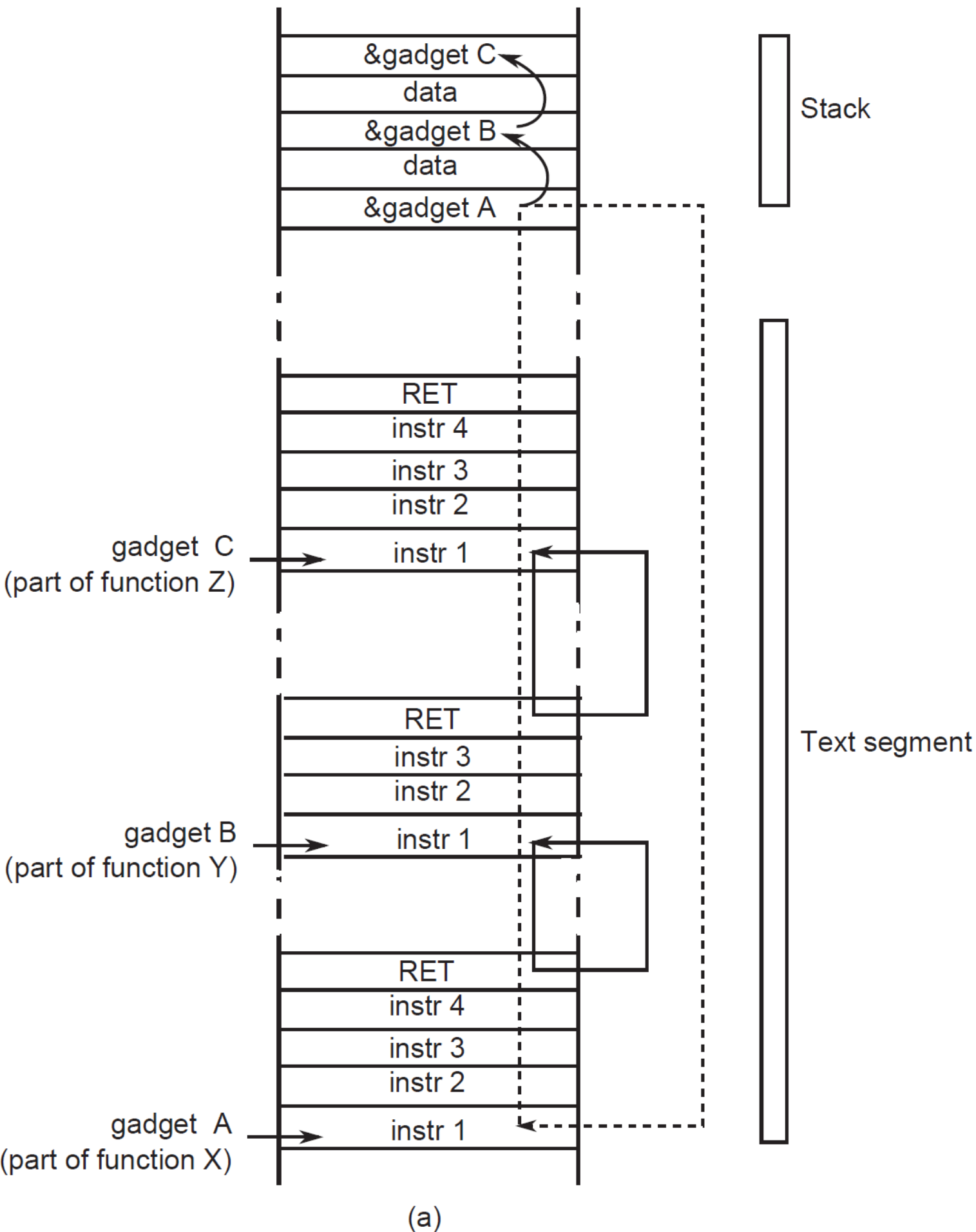


# DEP / NX / W $\oplus$ X

- Another defense is to try and make sure that even if an attacker can overflow a buffer and change the return address, they can't execute code
- In the previous example, attacker code was placed into a stack buffer
- So, simple solution: don't allow data to be executable!
- Generally this requires hardware support
  - NX bit in x86 page protections

# Return-Oriented Programming

- Despite DEP, attackers can still run code!
- Instead of trying to execute their *own* code, attackers can change the return address to point to existing code in memory
- By setting up values on the stack, they can bounce around executing tiny snippets of code ending in ret
- Thus, by chaining these together, *arbitrary* computation can be performed – without executing anything marked as data



# Dangling Pointers

- An increasingly common and exploited class of vulnerability is the *dangling pointer* or *use after free* vulnerability
- Scenario: programmer frees an object, but then continues to use the pointer afterward
- Problem: new allocations may use the same space, placing a *different* object at the same location
- If an attacker can manage to get his data placed into that freed space, can manipulate the program into executing his code

# Use After Free

```
01. int *A = (int *) malloc (128);           /* allocate space for 128 integers */
02. int year_of_birth = read_user_input (); /* read an integer from standard input */
03. if (input < 1900) {
04.     printf ("Error, year of birth should be greater than 1900 \n");
05.     free (A);
06. } else {
07.     ...
08.     /* do something interesting with array A */
09.     ...
10. }
11. ... /* many more statements, containing malloc and free */
12. A[0] = year_of_birth;
```

# TOCTOU

- One final class of attacks is a *time of check to time of use* attack
- These arise any time you have a situation where
  1. A security check on some object is made
  2. The program does something with that object
  3. An attacker can intervene in between 1 & 2 and cause it to operate on a different (attacker-controlled) object

# TOCTOU Example

```
int fd;  
if (access ("./my_document", W_OK) != 0) {  
    exit (1);  
fd = open ("./my_document", O_WRONLY)  
write (fd, user_input, sizeof (user_input));
```

# TOCTOU Attack

- To exploit this program, the attacker runs the program with a my\_document he controls
- After the access check is done, he deletes my\_document and creates a symbolic link from my\_document to some sensitive file
  - Getting the timing right here may be tricky!
- The program now opens and writes to my\_document, but that now results in writing to the sensitive file



# Preventing TOCTOU

- In general, the only way to prevent TOCTOU attacks is through careful API design
- In this case, we want to make sure that the file cannot be changed between the access check and the call to open
- One way to do this is to open the file and then perform access checks on the file descriptor (which can't be changed by the attacker) rather than the file (which can)