Crypto Part 1
(and Software Defenses Wrap-up)
CMSC 23200, Winter 2024, Lecture 4

Grant Ho and Blase Ur
University of Chicago
Assignment 2: Logistical Note

For Problem 4, your solution *cannot* involve executing any code placed on the stack!

- Currently: configuration error in the VMs that makes Target 4’s stack executable.
- However, grading will run your solution against Target 4 with a non-executable stack, so your solution cannot use any shellcode on the stack.
- **Advice:** Implement a return-to-libc attack (as per the instructions)
Outline: Crypto + Software Security Wrap-up

1. Memory Safety Defenses
   - Fuzzing
   - Memory Safe Languages

2. Crypto Part 1: Symmetric Key Cryptography
Program Fuzzing: Find bugs before release

Idea: Developer runs their program on huge number of automatically-generated inputs, searches for crashes, and fixes bugs before releasing software

Linux Mint fixes screensaver bypass discovered by two kids

Two children playing on their dad's computer accidentally found a way to bypass the screensaver and access locked systems.

"A few weeks ago, my kids wanted to hack my Linux desktop, so they typed and clicked everywhere while I was standing behind them looking at them play," wrote a user identifying themselves as robo2bobo.

According to the bug report, the two kids pressed random keys on both the physical and on-screen keyboards, which eventually led to a crash of the Linux Mint screensaver, allowing the two access to the desktop.

"I thought it was a unique incident, but they managed to do it a second time," the user added.
Two Types of Fuzzing Strategies

**Mutation-based (dumb):** Take an initial set of examples (program inputs) and make random changes to them.

- Millions of inputs (can run fuzzing forever)
- Possibly lower quality, unlikely to find certain bugs / types of inputs

**Generative (smart):** Describe inputs to fit format/protocol, then generate inputs from that grammar with changes.

- Run with fewer inputs, which can be directed to certain bug types or code logic
Problems with Fuzzing

**Mutation-based (dumb):** How long to run? And we need a strong server.

**Generative (smart):** Run out of test cases. A lot more work.

**General problems:**
- Need to identify when bug/crash occurs automatically.
- Don’t want to report same bug 1000s of times.
- How do we prioritize bugs?
Fuzzing in Production

AFL: Popular open-source fuzzer released by Google

Google/Microsoft constantly fuzz products with dedicated servers/VMS.

Anecdote: Found 95 vulnerabilities in Chrome during 2011.

OneFuzz

A self-hosted Fuzzing-As-A-Service platform

Project OneFuzz enables continuous developer-driven fuzzing to proactively harden software prior to release. With a single command, which can be baked into CI/CD, developers can launch fuzz jobs from a few virtual machines to thousands of cores.
Memory-Safe Languages

Many of our problems can be solved by using “memory-safe” languages.

- The programming model for these languages does not allow for such bugs (e.g., no access to pointers / mem addr’s and built-in object bounds checking).

<table>
<thead>
<tr>
<th>Not Memory-Safe</th>
<th>Memory Safe</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Java</td>
</tr>
<tr>
<td>C++</td>
<td>Python</td>
</tr>
<tr>
<td>Assembly</td>
<td>Javascript</td>
</tr>
<tr>
<td></td>
<td>Rust, Go, Haskell, …</td>
</tr>
</tbody>
</table>

Ideally, we’d avoid writing programs in unsafe languages, but lots of legacy code (and low-level stuff) are written in C/C++. 
Recap: Software Defenses

Pre-deployment, before the program runs: find or prevent bugs
- Fuzzing: proactively finding & fixing bugs by testing many program inputs
- Memory safe languages: automatically avoid exploitable memory bugs
- Done by the application developer

Program runtime: stopping exploits / violations of program’s memory
- Stack Canaries, ASLR, DEP/W+X, etc.
- Implemented by the compiler (stack canary) or operating system (ASLR, W+X)
- Attacks adapt & evolve (Stack reading, ROP attacks, etc.)

Post-exploitation (not covered): limit possible damage from compromise
- Sandboxing and VMs
- Done by user/admin of the system or the app developer (e.g., web browsers)
Cryptography: Part 1

(Slides adapted from David Cash and Dan Boneh)
Outline: Cryptography Part 1

1. Memory Safety Defenses
   - Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography
   - Common goals & Threat models
   - Encryption & Basic ciphers
   - One-time pads and Secure encryption
   - Stream ciphers
   - Message Authentication Codes (MACs)
What is Cryptography (for CMSC 23200)?

Cryptography develops algorithms that achieve security goals (CIA).

Cryptography involves using math / theory to stop adversaries.

This Course:
- A brief overview of major crypto concepts and tools
- Cover (some) big “gotchas” in crypto deployments
- Not going to cover math, proofs, or many theoretical details. Consider taking CS284 (Cryptography)!
Common High-Level Goal: Create a Secure Channel

Goal: Attacker does not learn anything about the contents of messages and cannot tamper with their contents.
Example 1: Secure communication
(protecting data in motion)

Unable to learn contents or tamper with data
Example 2: Protected files (protecting data at rest)

File system

Alice
T=0

File 1

File 2

Unable to learn contents or tamper with data (file)

Alice
T=1
Three Key Security Goals of Cryptography

1. **Confidentiality**: an attacker cannot learn the contents of our data

2. **Integrity**: an attacker cannot modify the contents of our data

3. **Authentication**: an attacker cannot masquerade as someone else, or make us believe their message/data was sent by someone else
## Four Cryptography Problems / Tools

<table>
<thead>
<tr>
<th>Security Goal</th>
<th>Confidentiality</th>
<th>Authenticity/Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-shared key?</td>
<td>Yes (&quot;Symmetric&quot;)</td>
<td>No (&quot;Asymmetric&quot;)</td>
</tr>
</tbody>
</table>
### Four Cryptography Problems / Tools

<table>
<thead>
<tr>
<th>Pre-shared key?</th>
<th>Security Goal</th>
<th>Confidentiality</th>
<th>Authenticity/Integrity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes (&quot;Symmetric&quot;)</td>
<td>Symmetric Encryption</td>
<td></td>
<td>Message Authentication Code (MAC)</td>
</tr>
<tr>
<td>No (&quot;Asymmetric&quot;)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Confidentiality** is achieved through *Symmetric Encryption* when there is a pre-shared key.
- **Authenticity/Integrity** is ensured with *Message Authentication Code (MAC)* when there is no pre-shared key.
Four Cryptography Problems / Tools

Pre-shared key?  | Confidentiality  | Authenticity/Integrity
---|---|---
Yes ("Symmetric")  | Symmetric Encryption  | Message Authentication Code (MAC)
No ("Asymmetric")  | Public-Key Encryption  | Digital Signatures
Outline: Cryptography Part 1

1. Memory Safety Defenses
   • Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography
   • Common goals & Threat models
   • Encryption & Basic ciphers
   • One-time pads and Secure encryption
   • Stream ciphers
   • Message Authentication Codes (MACs)
Ciphers (a.k.a. Symmetric Encryption)

A cipher is a pair of algorithms Encrypt, Decrypt:

- **Encryption** algorithm: Encrypt(K, m) = c
  - Convert a plaintext message \( m \), into an encrypted message \( c \) (ciphertext)
- **Decryption** algorithm: Decrypt(K, c) = m
  - Convert a ciphertext \( c \), back into its plaintext message \( m \)
Encryption: Providing Confidentiality

Threat Model: Passive attacker
- Adversary see the ciphertexts, but they cannot modify them in any way
- Attacker’s goal: learn something about plaintext messages from ciphertexts

Today’s Lecture: Symmetric key setting:
- Alice & Bob already have a shared secret key, $K$, that the attacker does not know
Ciphers (a.k.a. Symmetric Encryption)

A cipher is a pair of algorithms Encrypt, Decrypt:

Requirements of a Secure Cipher:

- **Correctness**: decryption recovers the same message.
  - Encrypt(K, m) = c and Decrypt(K, c) = m

- **Confidentiality (Security)**: the ciphertext c reveals nothing about the message m (other than the message length)
Historical Cipher: ROT13 (“Caesar cipher”) 

Encrypt(K,m): shift each letter of plaintext forward by K positions in the alphabet (wrap from Z to A).

Plaintext:   DEFGH
Key (shift):  2
Ciphertext:  FGHKL

Plaintext:   ATTACKATDAWN
Key (shift):  13
Ciphertext:  NGGNPXNGQNJAG
**Historical Cipher: Substitution Cipher**

**Encrypt(K,m):** The key K is a permutation $\pi$ on \{A,… Z\}. Apply $\pi$ to each character of m to create c

\[ M: \text{ATTACKATDAWN} \]
\[ K: \pi \]
\[ C: \text{ZKKZAMZKYZGT} \]

How many keys?
\[ 26! \approx 2^{88} \]
9 million years to try all keys at rate of 1 trillion/sec

**Q:** Is this secure?
Cryptanalysis of Substitution Cipher

Distribution of letters in English text is not uniform:

- Can guess letters in a long msg by computing their frequency
Outline: Cryptography Part 1

1. Memory Safety Defenses
   - Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography
   - Common goals & Threat models
   - Encryption & Basic ciphers
     - One-time pads and Secure encryption
   - Stream ciphers
   - Message Authentication Codes (MACs)
Quick recall: Bitwise-XOR operation

We will use bit-wise XOR:

\[
\begin{array}{c}
0101 \\
\oplus 1100 \\
\hline
1001
\end{array}
\]

Some Properties:

- \(X \oplus Y = Y \oplus X\)
- \(X \oplus X = 000\ldots0\)
- \(X \oplus Y \oplus X = Y\)
Cipher Example: One-Time Pad (OTP)

**Key** $K$: Bitstring of length $L$

**Plaintext** $M$: Bitstring of length $L$

<table>
<thead>
<tr>
<th>Encrypt($K,M$): Output $K \oplus M$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrypt($K,C$): Output $K \oplus C$</td>
<td></td>
</tr>
</tbody>
</table>

Example:

\[
\begin{align*}
0101 & \quad (K) \\
\oplus 1100 & \quad (M) \\
\hline
1001 & \quad (C)
\end{align*}
\]

Correctly decrypts because

\[
K \oplus C = K \oplus (K \oplus M) = (K \oplus K) \oplus M = M
\]

Q: Is the one-time pad secure?

Bigger Q: What does “secure” even mean?
Evaluating Security of Crypto Algorithms

Kerckhoff’s Principle:
Assume the adversary knows your algorithms and implementation. The only thing they don’t know is the key.

Example:
- Adversary knows you are running SSH, and they know logic/code of all the ciphers that SSH allows (e.g., by downloading the open-source software itself)
- But they do not know the keys that Alice & Bob use
The adversary sees ciphertexts and attempts to recover some “useful information” about plaintexts.

Other attack settings are important
(e.g. adversary can ask for some encryptions, some decryptions…)

Adversary Goal: Break Confidentiality
Partial Knowledge & Recovering Partial Information

- Recovering entire messages is useful
- But recovering **partial information** is also be useful & dangerous

A lot of information is missing here.

But can we say who this is?

- Attacker may know large parts of plaintext already (e.g. formatting strings or application content).
  The attacker tries to obtain something it doesn't already know.

  \[ M = \text{http://site.com?password=**********} \]
Secure Encryption Goal

An **attack** is successful as long as it recovers *any* new info about the plaintext that is useful to the adversary.

Encryption must hide *all possible partial information* about plaintexts, since what is useful or dangerous is situation-dependent.
Attacks can succeed without recovering the key

$K \rightarrow m_1, \ldots, m_q \rightarrow \text{Clever attacker} \rightarrow C_1, \ldots, C_q \rightarrow K \leftarrow m/\bot$

**Full break:** Adversary recovers $K$, decrypts all ciphertexts.

**However:** Clever attackers may learn plaintext information from ciphertexts without recovering the key. If so, the attack has succeeded / encryption algorithm is insecure.
Security of the One-Time Pad (OTP)

One-time pad: if an adversary sees \textbf{only one} ciphertext under a random key, then \textit{any} plaintext is equally likely, so they cannot recover any partial information \textit{besides the plaintext length}.

\[
\begin{align*}
\text{Ciphertext observed: } & 10111 \\
\text{Possible plaintext: } & 00101 \\
\Rightarrow \text{Possible key: } & 10010
\end{align*}
\]

1. Adversary goal: Learn partial information from plaintext
2. Adversary capability: Observe a single ciphertext
3. Adversary compute resources: Unlimited time/memory (!)
Issues with One-Time Pad (OTP)

1. Reusing a pad is insecure
2. One-Time Pad has a long key
Issue #1: Reusing a One-Time Pad is Insecure

\[
\text{HELLOALICE} \oplus \text{Pad (Key)} = C_1 \\
\text{HELLOALICE} \oplus \text{Pad} = \text{Pad}
\]

\[
\text{PWDHAMSTER} \oplus \text{Pad (Key)} = C_2 \\
\text{PWDHAMSTER} \oplus \text{Pad} = \text{Pad}
\]
Issue #1: Reusing a One-Time Pad is Insecure

Has led to real attacks:
- Project Venona (1940s) attack by US on Soviet encryption
- MS Windows NT protocol PPTP
- WEP (old WiFi encryption protocol)
- Fortiguard routers! [link]
Issue #2: One-Time Pad Needs a Long Key

By definition: OTP needs Key-length ≥ Plaintext-length
• Long message = long key required

In practice:
- Use *stream cipher*: Encrypt(K,m) = G(K)⊕m
- Use *nonces* to encrypt multiple messages
  (ensures that even if we send same msg twice, the ciphertext is different)
Outline: Cryptography Part 1

1. Memory Safety Defenses
   - Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography
   - Common goals & Threat models
   - Encryption & Basic ciphers
   - One-time pads and Secure encryption
   - Stream ciphers
   - Message Authentication Codes (MACs)
Key Idea: Given a random key, $K$, create an extremely large pseudo-random string that can be used as a one-time pad

- Cryptographic functions called pseudo-random number generators (PRNGs) that can do this
Tool to address key-length of OTP: Stream Ciphers

Stream cipher syntax: Algorithm $G$ that takes one (smaller) input and produces a very long bit-string as output.

Typically 16 or 32 bytes.

Key (Seed) $k$: 1100..11

$G(k)$: 111110100010001110101001000101100100111100...

$\oplus$ DONUTS DONUTS DONUTS DONUTS DONUTS DONUTS DONUTS DONUTS DONUTS DONUTS DONUTS

Use $G(key)$ as the one-time pad.
- Can now encrypt messages much longer than the key.
Security goal: When $k$ is random and unknown, $G(k)$ should “look” random.

... even to an adversary spending a lot of computation.

Much stronger requirement that “passes statistical tests”.

Brute force attack: Given $y = G(k)$, try all possible $k$ and see if you get the string $y$.

Clarified goal: When $k$ is random and unknown, $G(k)$ should “look” random to anyone who can’t run a brute force attack.

(key length = 256-bits is considered strong now)
Practical Stream Ciphers (Not covered in this class)


![Diagram of RC4](image)

**ChaCha20 (2007)**: Successfully deployed replacement. Supports *nonces*. 

![Diagram of ChaCha20](image)
Sending Multiple Messages w/ Stream Ciphers: Pad Reuse?

\[ m_1 \oplus G(k) \quad \text{ciphertext} \]

\[ m_2 \oplus G(k) \quad \text{ciphertext} \]

\[ \ldots \]
Addressing pad reuse: Stream cipher with a nonce

Stream cipher with a nonce: Algorithm G that takes **two inputs** and produces a very long bit-string as output.

- Nonce IV:
  - 1100..11
- Key/Seed k:
  - 1100..11

\[ G(\text{IV}, k) : \overline{11110100010001110101001000101100100111100...} \]

- “nonce” = “number once”.
- Usually denoted IV = “initialization vector”

Security goal: When \( k \) is random and unknown, \( G(\text{IV}, k) \) should “look” random and independent for each value of \( \text{IV} \).
Solution 1: Stream cipher with a nonce

- If nonce repeats, then pad repeats
Example of Pad Re-use: WEP

**IEEE 802.11b WEP:** WiFi security standard ’97-‘03

- IV

  IV is a 24-bit wide counter

- Repeats after $2^{24}$ frames (≈16 million)
- IV is often set to zero on power cycle

**Solutions:** (WPA2 replacement)
- Larger IV space, or force rekeying more often
- Set IV to combination of packet number, address, etc
Example of Pad Re-use: WEP

IEEE 802.11b WEP: WiFi security standard ’97-'03

Warning: Broken

**Serious flaw in WPA2 protocol lets attackers intercept passwords and much more**

KRACK attack is especially bad news for Android and Linux users.

DAN GOODIN - 10/15/2017, 11:37 PM

Solutions: (WPA2 replacement)

- Larger IV space
- Set IV to combination of packet number, address, etc

parameters to their initial values, KRACK forces the nonce reuse in a way that allows the encryption to be bypassed. Ars Technica IT editor Sean Callaghe has much more about KRACK here.
Issues with One-Time Pad

1. Reusing a pad is insecure  
   Use unique nonces

2. One-Time Pad needs a long key  
   Use stream cipher with short key
Outline: Cryptography Part 1

1. Memory Safety Defenses
   - Fuzzing and Memory Safe Languages

2. Symmetric Key Cryptography
   - Common goals & Threat models
   - Encryption & Basic ciphers
   - One-time pads and Secure encryption
   - Stream ciphers
   - Message Authentication Codes (MACs)
Integrity: Message Authentication Codes (MACs)

- Encryption provides confidentiality: a *passive* attacker can’t learn anything about the data we’re storing or using.

- **Integrity**: an (active) attacker cannot tamper with the data in an *undetectable* manner.
  - i.e., allows user to check if the data they received is exactly what was sent or if it has been modified.
Integrity: New Threat Model (Active Attacker)

Threat model: *Active attacker* that can tamper with communication

- Attacker not only sees all ciphertexts, but can also actively **modify** ciphertexts during transmission, **inject** their own data as additional “ciphertexts”, **reorder or delete** ciphertexts
- Often known as a Man-in-the-Middle (MITM) attacker
OTP & Stream Ciphers Do Not Provide Integrity

\[
PAYALICE$1 \oplus \text{Pad} = C \\
\oplus \text{000ALICE00} = C' \oplus \text{000DAVID00} = \text{PAYDAVID$1}
\]
Stream ciphers do not give integrity

M = please pay ben 20 bucks

C = b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e464a782871c2d

C' = b0595fafd05df4a7d8a04ced2d1ec800d2daed851ff509b3e464a782871c2d

M' = please pay ben 21 bucks

Encryption alone does not provide integrity
(fundamentally not designed to)
Providing Integrity: Message Authentication Code

**Idea:** Append a special tag to each message that (1) validates the message content (different msg = different tag) and (2) can only be computed if a user knows the secret key K.
Providing Integrity: Message Authentication Code

A **message authentication code (MAC)** is an algorithm that takes as input a key and a message, and outputs an “unpredictable” tag.

\[ D \rightarrow \text{MAC}_K (D) \rightarrow T \]

\[ K \]

Check: \[ T = \text{MAC}_K (D) \]?

\( D \) will usually be a ciphertext, but is often called a “message”. 
MAC Security Goal: Unforgeability

MAC satisfies **unforgeability** if it is infeasible for Adversary to fool Bob into accepting $D'$ and $T'$ as a valid (msg, MAC) pair, for a $D'$ that has not been previously seen.
MAC Security Goal: Unforgeability

D = please pay ben 20 bucks
T = 827851dc9cf0f92ddc552572ffd8bc

D', T' = baeaf48a891de588ce588f8535ef58b6

Unforgeability: Attacker cannot create T' for any new D'.

MACs do NOT need to provide any confidentiality (no encryption on this slide)
MACs In Practice: Use HMAC or Poly1305-AES


- Other, less-good option: AES-CBC-MAC (bug-prone)
Authenticated Encryption

Encryption that provides **confidentiality** and **integrity** is called **Authenticated Encryption**.

- Built using a good stream cipher and a MAC.
  - Ex: Salsa20 with HMAC-SHA2
- Best solution: Use ready-made Authenticated Encryption
  - Ex: AES-GCM is the standard
The End