Introduction to Cybersecurity

Cryptographic Protocols

Summary

Secure Communication Protocols
- Cryptographic Primitives & Protocols
- Reflection and Replay Attacks
- Challenge Response Handshakes
- Needham-Schroeder-Lowe Protocol
- SSL/TLS

Fuzzing
- What is Fuzzing?
- Dumb vs. Smart Fuzzing
- Determining Exploitability

An untrusted world
- Trusted principals exchange messages on a network populated by malicious entities
- Everybody can read and write the messages in transit on the network
Safe communication

- For securing a communication, we need to
  - identify the security goals
  - determine the threat model (which enemy do we have to defend ourselves from?)
  - protect the messages in transit on the network accordingly

Some security goals

- **Secrecy.** Only the authorized recipient should be able to learn the message
- **Integrity.** The recipient should be able to determine whether the message has been altered during transmission or not
- **Authenticity.** There exist two variants:
  - non-injective agreement: the recipient of an authentication request should be able to verify the identity of the requester and both should agree on their respective roles
  - injective agreement: same as above, plus the recipient should be able to verify the freshness of the authentication request

Some other security goals

- Unlinkability
- Accountability
- Trust
- Anonymity
- Receipt-freeness
- Coercion-resistance
- Non-repudiation
- Strong secrecy
- Secrecy against offline attacks
Threat Model: A Simple Example

- “Give Eve 1000$”

- “Give Eve 1000$” → “Give Eve 2000$”

  - Eve intercepts the first message and modifies it in order to get 2000$

Protecting Communication

- How to protect our network communication?

Cryptography!

Cryptographic Primitives

What we have learned:
- Symmetric Encryption
  - Streamciphers
  - Blockciphers: DES, AES
  - Modes of Operation
- MACs, Hashes
- Asymmetric Encryption
  - El-Gamal, RSA
- Digital Signatures

Cryptographic Primitives that provide
- Secrecy
- Integrity
- Authenticity
- ...
  for message transmissions over insecure channels
Cryptographic Protocols

- Use cryptographic primitives to achieve more complex security goals
- **Example: Safe Browsing**
  - Authenticate Server to Client
  - Exchange key material for secret communication
- Implemented in **SSL/TLS** using
  - Digital Signatures (to authenticate Server)
  - Public Key Encryption (to exchange key material)
  - Symmetric Encryption (for secret communication)
- But is simply adding cryptography enough?

Adding Cryptography

\[ \text{“Give Eve 1000$”} \]

- $k$ is a **symmetric key** shared between Alice and Bob: only Alice and Bob can encrypt and decrypt / sign and verify messages with $k$
- Symmetric cryptography protects the secrecy and the integrity of the message
  - The attacker cannot read or modify the transfer request
- Does this really prevent Eve from obtaining 2000$?

Attack I

- The attacker intercepts the transfer request and sends it back to Alice, who recognizes the message as generated by Bob and performs the transfer
  \[ \text{“Give Eve 1000$”} \]

- The symmetric nature of the encryption key $k$ does not allow Alice to verify whether the ciphertext has been generated by Bob or herself
Cryptography is not enough

- Unfortunately, cryptography is not enough!
- The attacker can circumvent cryptography and break the security goals of the protocol by:
  - intercepting,
  - duplicating,
  - sending back the messages without need to break the encryption scheme
- In the following, we assume that cryptography is a fully reliable black box and focus on how cryptography is used.

Attacker threats

- Attacks are often surprising and hard to predict,
- they can still be roughly classified according to the kind of interaction between the attacker and the protocol sessions.

Attack I

- The attacker intercepts the transfer request and sends it back to Alice, who recognizes the message as generated by Bob and performs the transfer:

  \[
  \rightarrow \{ \text{Give } B \text{ve 1000}\} \_a \rightarrow \]
  \[
  \leftarrow \{ \text{Give } B \text{ve 1000}\} \_k \rightarrow \]

- Reflection attack: an attack in which a message is sent back to its generator.
Avoiding Reflection Attacks

- A solution is to break the symmetry of the cryptographic scheme by inserting the originator’s identifier (or the intended receiver’s one)

\[ \{ \text{Alice, “Give Eve 1000$”} \} \]

Attack II: Replay Attack

- **Replay Attack**: The same message is duplicated and sent several times to the intended recipient

\[ \{ \text{Alice, “Give Eve 1000$”} \} \]

This protocol does not guarantee injective agreement (Eve still gets 2000$)

- A possible solution is to insert a timestamp \( t \) for guaranteeing the freshness of the message
  - The authentication request is accepted only if it has been recently generated and no authentication request with the same timestamp has previously been accepted
  - This involves a global clock

\[ \{ \text{Alice, “Give Eve 1000$”, t} \} \]
Attack II: Replay Attack

- Another solution is to exploit a challenge-response nonce handshake
  - A nonce is a randomly generated number n, used in a single protocol session and then discarded
  - An authentication request is accepted only if no authentication request with the same nonce has previously been accepted

\[ \{Alice, \text{“Give Eve 10008”, n}\} \]

Challenge-response Handshakes

- Challenge-response nonce handshakes are very common in cryptographic protocols and they can be implemented in different ways
  - PC (plain-cipher) handshakes: the challenge is in clear and the response is encrypted
  - CP (cipher-plain) handshakes: the challenge is encrypted and the response is in clear
  - CC (cipher-cipher) handshakes: both the challenge and the response are encrypted

- The common idea is that principals prove their identities by encrypting/decrypting the challenges and the responses
- The security properties provided by the three nonce handshakes are however different

PC Handshake

- Bob authenticates Alice sending message m
- This protocol is also known as ISO two-pass unilateral authentication protocol

\[ \{A, m, n\} \rightarrow \]

- The response might be signed with Alice's private key and the identifier A replaced by B.
- Question: Why do we have to change the identifiers?
- Answer: Otherwise, Bob does not know whom Alice is willing to authenticate with
CP Handshake

- Bob authenticates Alice receiving message \( m \)
  - The second message may be seen as a receipt acknowledgment
  - The challenge might be encrypted by Alice’s public key (and the identifier \( A \) replaced by \( B \))

\[
\begin{align*}
&\text{Bob} & \rightarrow \{A, m, n\}_{A, B} \\
&\text{Alice} & \leftarrow n
\end{align*}
\]

CC Handshake

- Bob authenticates Alice receiving message \( m_1 \) and sending message \( m_2 \)
  - The challenge might be encrypted with Alice’s public key and the response encrypted with Bob’s public key

\[
\begin{align*}
&\text{Bob} & \rightarrow \{B, m_1, n\}_{A, B} \\
&\text{Alice} & \leftarrow \{A, m_2, n\}_{A, B}
\end{align*}
\]

- Question: can we replace \( B \) with \( A \) in the first message (or \( A \) with \( B \) in the second one)?
- Answer: no, otherwise it would be possible to mount a reflection attack

Mutual Authentication Protocols

- Nonce handshakes can be combined in order to allow principals to authenticate each other
- This protocol is also known as ISO three-pass authentication protocol and it is composed of a PC handshake and a CC handshake

\[
\begin{align*}
&\text{Bob} & \rightarrow \{B, m_1, n_1, n_2\}_{A, B} \\
&\text{Alice} & \leftarrow \{m_2, n_1, n_2\}_{A, B}
\end{align*}
\]

- Question: can we remove Bob’s identifier from the second message?
- Answer: no, otherwise it would be possible to mount a reflection attack
Needham-Schroeder Protocol

- \(\text{pk}(k_A)\) and \(\text{pk}(k_B)\) are Alice and Bob’s public keys, respectively
- This protocol was proposed in ’78

\[
\begin{align*}
\{B, n_B\}_{\text{pk}(k_A)} &
\{n_B, n_A\}_{\text{pk}(k_B)} \\
\{n_A\}_{\text{pk}(k_A)} &
\{n_A\}_{\text{pk}(k_A)}
\end{align*}
\]

- The aim is to guarantee the secrecy and the authenticity of the two nonces, which are then used for generating a symmetric session-key shared between Alice and Bob
- All messages are encrypted, two CC handshakes

Attack III

- Unfortunately, this protocol is not secure (Gavin Lowe ’86)

\[
\begin{align*}
\{B, n_B\}_{\text{pk}(k_E)} &
\{B, n_B\}_{\text{pk}(k_A)} \\
\{n_B, n_A\}_{\text{pk}(k_E)} &
\{n_B, n_A\}_{\text{pk}(k_B)} \\
\{n_A\}_{\text{pk}(k_A)} &
\{n_A\}_{\text{pk}(k_A)}
\end{align*}
\]

- A believes that B is authenticating with her, while B is authenticating with E
- In the end, E learns the two nonces and can build the session key that A uses to talk with B

- Man-in-the-middle attack (MITM). The adversary E steps into the communication path and simply relays (possibly without changing) the messages between legitimate parties A and B, itself acting as a part of the communication
Needham-Schroeder-Lowe Protocol

- The fix proposed by Lowe consists in adding Alice’s identifier in the second ciphertext

\[ \{B, n_B\}_{pk(B)} \]

- B rejects the second ciphertext, as it does not come from E

\[ \{A, n_B, n_A\}_{pk(B)} \]

Outlook: How to Analyze Security Protocols?

Cryptographic Approach

- Strong guarantees for cryptographic primitives
- Error-prone for complex security protocols

Outlook: Verification of Abstracted Protocols

Analysis via algebraic abstractions of cryptography:

- "An encryption can only be processed by decrypting it, and only if the corresponding key is known"
- Algebraic structure + explicit set of rules \(\rightarrow\) automated protocol analysis possible
Outlook: How to Analyze Security Protocols?

**Cryptographic Approach**
- Strong guarantees for cryptographic primitives
- Error-prone for complex security protocols

**Algebraic Approach**
- Explicit list of logical deduction rules that the adversary must adhere to
- "Adversary: can do everything, if it is efficient"

**Cryptographic Approach**
- Requires paper&pencil math: complexity theory, probability theory
- Conceptual flaws in protocol design
- Cryptographic breaches
- Implementation mistakes

**Algebraic Approach**
- Cryptographic approach: algebraic approach: automata, deduction, automated reasoning
- Trustworthy analysis even for complex protocols

Cryptographic Protocols in Practice

- Tons of attacks (never ending list!)
  - Needham-Schroeder (1996)
  - Microsoft Passport (2001)
  - Public key Kerberos (2006)
  - DAA (2007,2008)
  - French Electronic Passport (2010)
  - 802.IIi WEP (2001)
  - SSL (2001,2009)
  - iSAXMP (2005)

- Flaws hard to spot, proofs hard to get right
What is SSL / TLS?

- Transport Layer Security protocol, version 1.2
  - De facto standard for Internet security
  - “The primary goal of the TLS protocol is to provide privacy and data integrity between two communicating applications”
    - Security against active, man-in-the-middle network attacker
  - Used to protect information transmitted between browsers and Web servers, VoIP, many other scenarios
- Based on Secure Sockets Layers protocol, ver 3.0
  - Same protocol design, different algorithms
- Deployed in nearly every Web browser

SSL / TLS in the Real World

History of the Protocol

- SSL 1.0
  - Internal Netscape design, early 1994
  - Lost in the mists of time
- SSL 2.0
  - Published by Netscape, November 1994
  - Several weaknesses
- SSL 3.0
  - Designed by Netscape and Paul Kocher, November 1996
- TLS 1.0
  - Internet standard based on SSL 3.0, January 1999
  - Not interoperable with SSL 3.0
  - TLS uses HMAC instead of MAC, can run on any port
- TLS 1.1 – April 2006
  - Protection against CBC attacks
- TLS 1.2 – August 2008 (current version)
  - MD5/SHA1 digests replaced
  - March 2011: removed backward compatibility with SSL
- TLS 1.3 DRAFT – July 2016
TLS Basics

- TLS consists of two protocols:
  - Handshake protocol
    - Use public-key cryptography to establish a shared secret key between the client and the server
  - Record protocol
    - Use the secret key established in the handshake protocol to protect communication between the client and the server
  - We will focus on the handshake protocol

TLS Handshake Protocol

- Two parties: client and server

Steps of the protocol:
1. Negotiate version of the protocol and the set of cryptographic algorithms to be used
   - Interoperability between different implementations of the protocol
2. Authenticate server and client (optional)
   - Use digital certificates to learn each other’s public keys and verify each other’s identity
3. Use public keys to establish a shared secret
ClientHello

Client announces (in plaintext):
- Protocol version she is running
- Cryptographic algorithms she supports
- A fresh, random number

ServerHello

Server responds (in plaintext) with:
- Highest protocol version supported by both client and server
- Strongest cryptographic suite selected from those offered by the client
- A fresh, random number
ServerHello

Client

Server responds (in plaintext) with:
- Highest protocol version supported by both client and server
- Strongest cryptographic suite selected from those offered by the client
- A fresh, random number

ServerKeyExchange

Server sends his public-key certificate containing either his RSA, or his Diffie-Hellman public key (depending on chosen crypto suite)

ServerKeyExchange

Server sends his public-key certificate containing either his RSA, or his Diffie-Hellman public key (depending on chosen crypto suite)
Client generates some secret key material and sends it to the server encrypted with the server's public key (if using RSA).

If the protocol is correct, C and S share some secret key material (\texttt{secret}) at this point.
Version Rollback Attack

Client

Server

C and S end up communicating using SSL 2.0 (weaker earlier version of the protocol that does not include "Finished" messages)

“Chosen-Protocol” Attacks

- Why do people release new versions of security protocols?
  - Because the old version got broken!
- New version must be backward-compatible
  - Not everybody upgrades right away
- Attacker can fool someone into using the old, broken version and exploit known vulnerability
  - Similar: fool victim into using weak crypto algorithms
- Defense is hard: must authenticate version early
- Many protocols had “version rollback” attacks
  - SSL, SSH, GSM (cell phones)

Version Check in TLS 1.0 (SSL 3.1)

Client

Server

If the protocol is correct, C and S share some secret key material (secret) at this point

Switch to key derived from secret, \( N_C, N_S \)

Check that received version is equal to the version in ClientHello

“Embed” version number into secret

Check that received version is equal to the version in ClientHello

Switch to key derived from secret, \( N_C, N_S \)
Conclusion

- Designing cryptographic protocols is highly error-prone, even for security experts.
- Novel cryptographic protocols often required in a number of settings (e.g., social networks, cloud computing, etc.).
- Implementing cryptographic protocols equally error-prone:
  - be careful with optimizations and simplifications!
- In the last years, security researchers developed a number of push-button tools for protocol analysis.
  - These tools are based on:
    - Mathematical models of cryptographic protocols
    - Formal methods (e.g., type systems, theorem proving) for the verification of security properties on these models.

Summary

Secure Communication Protocols

- Cryptographic Primitives & Protocols
- Reflection and Replay Attacks
- Challenge Response Handshakes
- Needham-Schroeder-Lowe Protocol
- SSL/TLS

Fuzzing

- What is Fuzzing?
- Dumb vs. Smart Fuzzing
- Determining Exploitability
What is fuzzing?

- Feed target automatically generated malformed data designed to trigger implementation flaws
- Fuzzer is the programmatic construct to do this
- Fuzzing framework typically includes library code to:
  - Generate fuzzed data
  - Deliver test cases
  - Monitor the target
- Publicly available fuzzing frameworks:
  - Spike, Peach Fuzz, Sulley, Schemer, OSS-Fuzz
- Requirement of Microsoft's Secure Development Lifecycle program
- Still a long way to go - many vendors do no fuzzing!

What data can be fuzzed?

- Virtually anything!
- Basic types:
  - bit, byte
  - word, dword, qword
- Common language specific types:
  - strings
  - structs
  - arrays
- High level data representations:
  - text
  - xml

Where can data be fuzzed?

- Across any security boundary, e.g.:
  - An RPC interface on a remote/local machine
  - HTTP responses & HTML content served to a browser
  - Any file format, e.g. Office document
  - Data in a shared section
  - Parameters to a system call between user and kernel mode
  - HTTP requests sent to a web server
  - File system metadata
  - ActiveX methods
  - Arguments to SUID binaries
What does fuzzed data consist of?

- Fuzzing at the type level:
  - Long strings, strings containing special characters, format strings
  - Boundary case byte, word, dword, qword values
  - Random fuzzing of data buffers
- Fuzzing at the sequence level
  - Fuzzing types within sequences
  - Nesting sequences a large number of times
  - Adding and removing sequences
  - Random combinations
- Always record the random seed!!

When to fuzz?

- Fuzzing typically finds implementation flaws, e.g.:
  - Memory corruption in native code
    - Stack and heap buffer overflows
    - Un-validated pointer arithmetic (attacker controlled offset)
    - Integer overflows
    - Resource exhaustion (disk, CPU, memory)
  - Unhandled exceptions in managed code
    - Format exceptions (e.g. parsing unexpected types)
    - Memory exceptions
    - Null reference exceptions
  - Injection in web applications
    - SQL injection against backend database
    - LDAP injection
    - HTML injection (Cross-site scripting)
    - Code injection

When not to fuzz

- Fuzzing typically does not find logic flaws
  - Malformed data likely to lead to crashes, not logic flaws
  - e.g. Missing authentication / authorization checks
- Fuzzing does not find design/repurposing flaws
  - e.g. A sitelocked ActiveX control with a method named "RunCmd".
  - However transitions in a state machine can be fuzzed...
    - Send well-formed requests out of order
    - But how to know when you’ve found a bug?
Two Approaches to Fuzzing

“Dumb”
- Fuzzer lacks contextual informational about data it is manipulating
- May produce totally invalid test cases
- Up and running fast
- Find simple issues in poor quality code bases

“Smart”
- Fuzzer is context-aware
  - Can handle relations between entities, e.g. block header lengths, CRCs
- Produces partially well-formed test cases
- Time consuming to create
  - What if protocol is proprietary?
- Can find complex issues

Pseudo-code for dumb fuzzer

```c
for each {byte|word|dword|qword} aligned location in file
  for each bad_value in bad_valueset
    file[location] := bad_value
    deliver_test_case()
```

Sample Config for Smatf Fuzzer: Config. Language

```c
...«_jpeg = AppendObject[type=JPEG, mode=OBJECT, length=0, data='"\x00\x00\x00\x00']" // JPEG
  AppendObject[type=JPEG, mode=OBJECT, length=0, data='\x00\x00\x00\x00']" // JPEG
  AppendObject[type=JPEG, mode=OBJECT, length=0, data='\x00\x00\x00\x00']" // JPEG
  AppendObject[type=JPEG, mode=OBJECT, length=0, data='\x00\x00\x00\x00']" // JPEG
...```

Foundations of Cybersecurity 2016
Sample config for smart fuzzer: XML fuzzer

Two approaches cont.

- Which approach is better?
- Depends on:
  - Time: how long to develop and run fuzzer
  - Security Code quality of target
  - Amount of validation performed by target
    - Can patch out CRC check to allow dumb fuzzing
  - Complexity of relations between entities in data format
- Don't rule out either!
  - Probably best approach: get a dumb fuzzer working first
  - Run it while you work on a smart fuzzer

Fuzzing in practice: the basic steps
Monitoring the target

1. Attach a debugger
   - Leverage existing functionality
   - Scripting, logging, crash dumps etc.

Monitoring the target

2. Write your own debugger
   - Actually easy to do
   - Lightweight, fast, full control

```cpp
BOOL WINAPI WaitForDebugEvent(
    __out  LPDEBUG_EVENT lpDebugEvent,
    __in   DWORD dwMilliseconds
);```

```typedef struct _DEBUG_EVENT { /* de */
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcess;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;```

Monitoring the target

3. Monitor resources:
   - File, registry, memory, CPU, logs
Deliver the test case

1. Standalone test harness
   - E.g. to launch to client application and have it load fuzzed file format
2. Instrumented client
   - Inject function hooking code into target client
   - Intercept data and substitute with fuzzed data
   - Useful if:
     - State machine is complex
     - Data is encoded in a non-standard format
     - Data is signed or encrypted

Determining exploitability

How to achieve exploitability?

- This process requires experience of debugging security issues, but some steps can be taken to gain a good idea of how exploitable an issue is...

A) Look for any cases where data is written to a controllable address – this is key to controlling code execution and the majority of such conditions will be exploitable

B) Verify whether any registers have been overwritten, if they do not contain part data sent from the fuzzer, step back in the disassembly to try and find where the data came from.

C) If the register data is controllable, point the register which caused the crash to a page of memory which is empty, fill that page with data (e.g., ‘aaaaa...’)

D) Repeat and step through each operation, until another crash occurs, reviewing all branch conditions which are controlled by data at the location of the (modified) register to ensure that they are executed
Determining exploitability

- Are saved return address/stack variables overwritten?
- Is the crash in a heap management function?
- Are the processor registers derived from data sent by the fuzzer (e.g. Ox61616161)?
- Is the crash triggered by a read operation?
- Can we craft a test case to avoid this?
- Is the crash triggered by a write operation?
- Do we have full or partial control of the faulting address?
- Do we have full or partial control of the written value?

Summary

Secure Communication Protocols

- Cryptographic Primitives & Protocols
- Reflection and Replay Attacks
- Challenge Response Handshakes
- Needham-Schroeder-Lowe Protocol
- SSL/TLS

Fuzzing

- What is Fuzzing?
- Dumb vs. Smart Fuzzing
- Determining Exploitability