CS 161

Computer Security

Notes

Druv Pai

Contents

1 Introduction

Security is defined in terms of an adversary. Essentially, it concerns methods to enforce a desired property (data confidentiality, user privacy, data and computation integrity, authentication, availability, etc) in the presence of an adversary.

The course structure follows the outline:

- Introduction to security (memory safety, operating system principles)
- Cryptography
- Network security
- Web security
- Miscellaneous topics, case studies

In this class, the goal is essentially to learn:

- How to think adversarially about computer systems
- How to build programs and systems with robust security properties

The first midterm is February something; the second midterm is April 1.

2 Security Principles

The first principle of a secure system is the attackers. We want to know who's attacking the system, and the best security is often to attack the attacker's motivation.

One thing we talk a lot in security about is a threat model. It doesn't make sense to say that a system is secure, or insecure. Any system is likely to be hackable by someone, given enough effort. But we don't really care about advanced hacking teams sponsored by foreign nations, if we know that they won't come after us. This allows us to move from the idea of "perfect" security to "good enough" security, and therefore the security problem is tractable.

The other side of the "people" aspect is the users. If a security system is unusable by the users, then it will be unused or circumvented by the users, and then it's worthless. An alternate method to break the system, even if the user uses the security, is to employ social engineering and obtain a legitimate key to the system. Often we blame a user when an atacker takes advantage of them, but if this is a recurrent problem then we've built a system that encourages users to do the wrong thing, so this is our problem.

Security often comes down to money. We don't put a $10 lock on a $1 item, unless the attacker can leverage that $1 item to attack something more important. The attacker doesn't risk exposing a $1,000,000 zero-day exploit on a random person. In this way, cost/benefit analyses appear all throughout security.

What countermeasures do we have against attacks?

- One approach is prevention; we want to stop any "bad thing" from happening at all. On one hand, if prevention works, it's great (for example, using memory-safe languages like Python makes you immune to a buffer overflow exploits). On theo ther hand, if prevention fails, it can fail hard (for example: bitcoin thefts, which cannot be reversed).
- Another approach is detection and response; detection sees whether something goes wrong, and response fixes that wrong, either reversing the issue or fixing the issue for the future. Responding to

false positives are not free, and if there are too many false positives, the detector is bad and wil not be paid attention to. On the other hand, false negatives are a complete failure.
- Another approach is defense in depth; if we do not have a perfect defense, we can layer multiple types of protection together. The attacker needs to breach all the defenses to gain access. But defense in depth isn't free, since we throw more resources. We can analyze the composition of detectors for this approach. The best case is when the two detectors are independent, and one can either detect overall when either detector flags the item, or when both detectors flag the item.
- Another approach is password authentication; an issue of this is that people have a hard time remembering multiple strong passwords, so they are often reused on multiple sites, and a security breach on any site compromises users on many different sites. Two ways to fix this are password managers, which require a single password and unlock all of the individual-site passwords, and two-factor authentication, which require both a correct password and separate device to access the account.

We want to keep data in "safes", where we want the contents to be inaccessible to an attacker. We want to measure how much time and capabilities needed for an attacker. In real life, safes have different ratings showing how much time a safecracking expert needs to break in.

Definition 1. The principle of least privileges means that programs should only have the permissions that they need to do their legitimate task.

This way, if a program is compromised, then the consequences are minimized.

One of the approaches to assessing security of a system design is to identify the trusted computing base (TCB), or the components does the security rely upon. Security requires that the TCB is correct, complete (can't be bypassed), and is itself secure (can't be tampered with). One powerful design approach is to do privilege separation, and keep privileged operation access to as small of a group as possible.

Example 2. The Chrome browser implements privilege separation. It is composed of the browser kernel, which is simple, trusted and cannot be attacked by vulnerabilities, so it is the TCB. The rendering engine is very complex and thus is placed in a sandbox.

Definition 3. To secure an access to some capability or resource, we construct a reference monitor, or a single point through which all access must occur (i.e. a network firewall).

The reference monitor should be un-bypassable ("complete mediation"), tamper-proof (is itself secure), and verifiable (correct) – so similar conditions to the TCB. One subtle form of reference monitor flaw concerns race conditions. Some more security principles are:

- Use fail-safe defaults; don't write cascading security breaches.
- Consider human factors; delegate human power to tasks humans are best at, and same for computers.
- A security system is only as secure as the weakest link.
- Don't rely on security through obscurity.
- Trusted path; build in trustworthy ways to interact with the system that can't be spoofed.

There is a vulnerability centering around time of check to time of use, whereby we induce a race condition. Imagine the pseudocode:

```
def withdraw(user, w):
    # contact central server to get balance
    b = balance(user)
```

```
    if b < w:
        break

    // contact server to set balance
    balance = b - w

    dispense(w)
```

If an attacker suspends the call after the check `b < w`, then they can call another `withdraw` function and steal `w` dollars. This is basically a race condition, and so one way to fix this is to set a lock around the procedure.

3 Buffer Overflow

A buffer overflow is when we write over the boundary of buffered memory.

```
char name[20];

void vulnerable() {
    ...
    gets(name);
    ...
}
```

If `name.size() > 19`, then we get a buffer overflow. we can think of this as an error for when array out of bounds.

If the code looks like

```
char name[20];
char instrux[80] = "none";

void vulnerable() {
    ...
    gets(name);
    ...
}
```

then the characters after `name` will change instruction code, which can mess things up.

For security, the issue can manifest as

```
char name[20];
int authenticated = 0;

void vulnerable() {
    ...
    gets(name);
```

```
    ...
}
```

Putting in 19 characters of name, a null byte, and then 1 will set the authenticated flag as 1, which is bad.

Another issue is

```
char line[512];
char command[] = "/usr/bin/finger";

void vulnerable() {
    ...
    gets(line);
    runv(command);
    ...
}
```

allows for arbitrary program execution via buffer overflow.

Another issue is

```
char name[20];
int (*fnptr) ();

void vulnerable() {
    ...
    gets(line);
    ...
}
```

whence we can use the buffer overflow to point the function pointer to anything, allowing you to call any function in the code, which makes it possible to run any secure or privileged operation. One can also introduce a new function at a known address via the same buffer overflow, allowing for arbitrary code execution.

Buffer overflows can also occur via stack-instantiated memory:

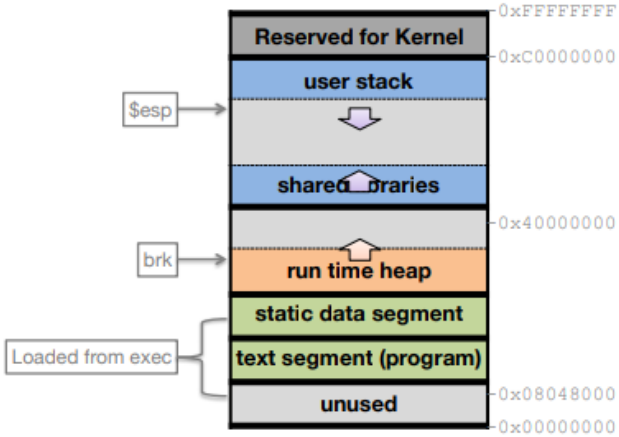
```
void vulnerable() {
    char buf[20];
    ...
    gets(buf);
    ...
}
```

We can even cross memory locations (stack, heap, etc.) using buffer overflows.

Memory layout will go, from top-down:

- User stack, including stack pointer, grows down
- Shared libraries, growing up
- Heap, growing up
- Static data segment

- Text segment



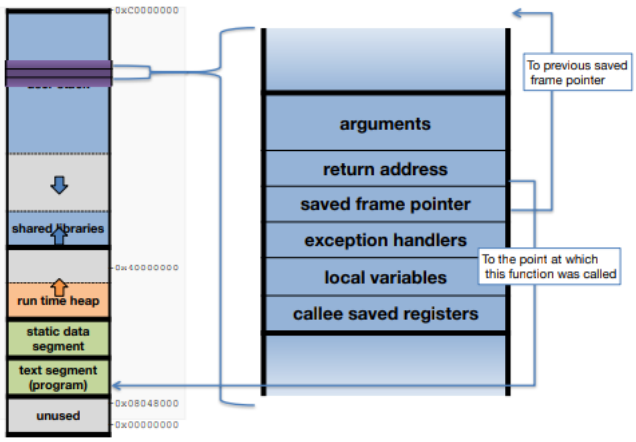
The main x86 registers are:

- `EAX-EDX`, the general purpose registers
- the `EBP` (frame pointer), which points to the start of the current call frame on the stack
- and `ESP` (stack pointer), which points to the current stack, and has the operations `push` (decrementing the stack pointer and pushing something there), and `pop` (loads something and increments the stack pointer)

The way that function calls work is :

- Place the arguments on the stack
- `call` the function, which `push`es the return address on the stack (`RIP` = Return Instruction Pointer)
- Function saves the old `EBP` on the stack (`SFP` = Saved Frame Pointer)
- Function does something
- Function restores everything, reloading `EBP`, `pop`ping `ESP` as necessary
- `RET` jumps to the return address that is currently pointed to by `ESP`, and can optionally `pop` the stack a lot further

The stack looks like this:



The buffer overflows can traverse and overwrite more of the stack than has been allocated. This means that the attacker can stash some malicious code at a memory address, then do a buffer overflow attack, rewriting the return address `RIP` of the function to point to the malicious code, then run this code. The easiest way to do this is to include the malicious code inside the buffer. If the code is very long, then the buffer can load a long (overflow) string that starts with innocuous text, then rewrites the return address to point to the malicious code, then include the malicious code.

How do we make code safe against buffer overflows? We can use the `fgets` API for C. The following code will write `64` bytes into a buffer that is `64` bytes long:

```
char buf[64];

void vulnerable() {
    ...
    fgets(name, sizeof(buf), stdin);
    ...
}
```

However, we must be careful. The following code looks safe, but is vulnerable:

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

We must be careful. The signature for `memcpy` is `memcpy(void *s1, const void *s2, size_t n);` which means that `n` is a variable of `size_t` type, so all `len` calls are interpreted as an `unsigned int`. As a result, if `len < 0` then the `memcpy` will write a lot of data since negative numbers are interpreted as large positive numbers in the unsigned representation. This is called a signed-unsigned bug.

The easy fix is to change the signature of `vulnerable`, that is,

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

Let's look at the following segment:

```
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';
}
```

Note that this can overflow, so `len = pow(2, 32)` or `len = pow(2, 32) - 1` will make the buffer `1` or `bytes` while writing `pow(2, 32)` or `pow(2, 32) - 1` bits via `memcpy`. This is an integer overflow bug.

Printing via `printf` is also questionable. A call for `printf("%d", x)` will place `x` above the string `"%d"` in the stack, then when evaluating the call it will look for the first integer above the string `"%d"` in the stack. This can be broken:

- `printf("100% dude!");` prints the value 4 bytes above the return address as an integer.
- `printf("100% sir!");` prints the bytes pointed to by the stack entry 4 bytes above the return address, until hitting `\0`.
- `printf("%d %d %d ... ");` will print the stack entries above the return address, as integers
- `printf("%d %s");` will print the stack entry 4 bytes above return address, then the bytes pointed to by the preceding entry (until `\0`).
- `printf("100% nuke em!");` writes to memory.

Note that `%n` writes the number of characters printed so far into the corresponding format argument (which should be a memory address). This is bad, because hackers can execute arbitrary code. The way to fix this is to not allow adversaries access to the format string.

The stack isn't the only segment that can be attacked; in fact, any segment can be attacked. For example, you can cause a lot of havoc by screwing with the heap, which is a good attack vector in C++.

If we can overwrite a `vtable` pointer (which attaches objects in C++ to methods and variables), then it's easy to make an object with methods that do anything that you want. The difference from rewriting the return address is that instead of overwriting with a pointer, we overwrite with a pointer to a table of pointers.

Heap overflows also exist, in the same fashion of stack overflows.

A use-after-free exploit is when an object is deallocated too early. Then the attacker writes new data in a newly allocated block that overwrites the `vtable` pointer, and then invokes the object.

Exploits are very brittle, and don't often work when ported to systems. Making an exploit robust is an art unto itself. A hack that helps is to create a `noop` sled. We don't just overwrite the pointer and then provide the code that we wish to execute. We can start the malicious code with many `noop` opeartions. This way, if we are a little off, it doesn't matter, since if we are close enough, the control flow will land in the stream of `noop`s and start running.

Some languages implement memory safety, which means that they ensure that programs do not access undefined memory. By access, we mean that attackers cannot read, write, or execute (transfer control flow) to memory that they aren't supposed to. This requires the language to prevent access to out-of-bounds memory, and prevent access to objects before or after the lifetime of the object. The main approaches for ensuring memory safety are

- Use a memory-safe language ("safe by design")
- Use a non-memory-safe language, and check bounds in your code
- Use a non-memory-safe language, and harden the code against common exploits

How do we have confidence that our code executes in a memory-safe (and correct, ideally) fashion? We build

up confidence on a function-by-function/module-by-module basis. The modularity provides boundaries for our reasoning:

- Preconditions impose constraints on the inputs and state before the function runs
- Postconditions impose constraints on the effect of the function

These describe a contract for using the function or module. These notions also apply to individual statements, and so statement i's postcondition should imply statement $(i+1)$'s precondition. In particular, invariants are conditions that always hold at a given point in a function.

An example of precondition is

```
/* requires: p != NULL (and p a valid pointer) */
int deref(int *p) {
    return *p;
}
```

An example of postcondition is

```
/* ensures: retval != NULL(and a valid pointer)*/
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

From now on the valid pointer condition is implicit.

Consider the following segment:

```
int sum(int a[], size_t n) {
    int total = 0;]
    for (size_t i=0; i<n; i++)
        total += a[i];
    return total;
}
```

If we make it memory safe, we can add the conditions

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
    int total = 0;]
    for (size_t i=0; i<n; i++)
        /* invariant: a != NULL && i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
    return total;
}
```

A more complicated loop might need us to use induction. For example, looking through a hash table is bad.

```c
char *tb1[N];

/* ensures: 0 <= retval < N */
unsigned int hash(char *s) {
    unsigned int h = 17; /* 0 <= h */
    while (*s) /* 0 <= h */
        h = 257*h + (*s++) + 3; /* 0 <= h */
    return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
    unsigned int i = hash(s);
    return tb1[i] && (strcmp(tb1[i], s) == 0);
}
```

Now, let's talk about hardening the code against attackers. We wish to prevent a program from having errors that are exploitable for code injection.

We now discuss stack canaries. The goal is to prevent the return pointer from being overwritten by a stack buffer. We wish to store the (randomly generated) canary before the saved return address. The function prologue pushes the canary, and the epilogue checks the canary against a stored value to see if it has changed (if it has, then shut down the program).

This is weak in several ways:

- We can learn the value of the canary, and overwrite it with itself
- Random-access write past the canary
- Overflow in the heap
- Overwrite function pointer or C++ object on the stack

Overall, this defense is bypassable but raises the bar for security attackers.

Another defense is to use non-executable pages (aka DEP, or $W\hat{X}$). Each page of memory has separate access permissions. The defense is to mark writeable pages as non-executable, so now we can't write code to the stack or heap. There's no noticeable performance impact of this method. We can attack this method by returning into `libc`, by setting up the stack and returning to `exec()`. In particular, we overwrite stuff above the saved return address with a "fake call stack", and overwrites the saved return address to point to the beginning of the `exec()` function. This is easy on x86 since arguments are passed on the stack. We can use return-oriented-programming to beat this. We can chain together the "return-to-`libc`" idea many times; in particular, we find a set of short code fragments (Gadgets) that when called in sequence execute the desired function. Then we inject into memory a sequence of saved "return addresess" that will invoke them. We can find enough gadgets scattered around existing code that they're Turing-complete, so we can compile a malicious payload to a sequence of these gadgets, using a ROP compiler.

Address space layout randomization randomly positions the base address of an executable and the positions of libraries, stack, and heap. ASLR and DEP is very strong. To break it, we need one vulnerability where the attacker can read memory. Just a single pointer to a known library will do, however the return address off the stack is often a great candidate, or a `vtable` for an object of a known type. Armed with this, the attacker can create a ROP chain, if a vulnerability exists there.

Automated testing is surprisingly effective at finding memory-safety vulnerabilities. When we have found a

problem, the program crashes. We can generate test cases in multiple ways:

- Random testing, where we generate random inputs
- Mutation testing, where we start frorm valid inputs and randomly flip bits in them
- Coverage-guided mutation testing, where we start from valid inputs, flip bits in the inputs, measure the coverage of each modification, and keep any inputs that covered new code.

4 Cryptography

Usually we discuss Alice sending a message to Bob, with the adversary (Eve the eavesdropper, or Malice the modifier) also present. Sometimes there's a third neutral party, Chris, depending on the protocol.

The main goals of cryptography are:

- Confidentiality: preventing adversaries from reading our private data
- Integrity: preventing attackers from altering some data
- Authenticity: ensuring that the expected user actually created some data

We study both symmetric-key cryptography and public-key (asymmetric-key) cryptography. In symmetric-key cryptography, the same secret key is used by both endpoints of a communication. In pubic-key cryptography, each party has a different key.

Kerkhoff's principle says that cryptosystems should remain secure even when the attacker knows all internal details of the algorithm or system. The key should be the only thing kept secret, and the system should be designed to make it easy to change keys that are leaked (or suspected to be leaked).

Symmetric Key Cryptography

Symmetric-key cryptography works like this. Alice has a key K and message M, and Bob also has K. Alice encrypts her message with the function $E_K(M)$, and sends it to Bob. If Eve intercepts the message, she gets garbage.

The symmetric-key encryption scheme consists of three algorithms:

- $G() = K$, which returns the key K.
- $E_K(M) = C$, which returns the encrypted message C.
- $D_K(C) = M$, which returns the original message M.

The scheme scrambles the information, but reveals the size of the message. Also, all messages must be of a fixed length, which is very inconvenient.

There's also a correctness property for each scheme. It says that $D_K(E_K(M)) = M$ for every K and M.

The main property we need to consider for each scheme is the security. We assume that the adversary sees the algorithms for G, E_K, and D_K, but does not know the key K generated.

A different, weaker definition is that no adversary can reconstruct M from a captured ciphertext C. The reason this is insecure is that a "secure" scheme can reveal all but one character of the ciphertext, or can reveal enough information such that the rest can be inferred.

The goal of such a scheme is therefore that no partial information about M may leak. Formally, no adversary should be able to distinguish two messages based on the encryption scheme.

We ensure that the scheme follows the goal by using a security game called IND-CPA.

Algorithm 1 IND-CPA Security Game

Input: Adversary $\mathcal{A}$ with guessing function A, randomly generated symmetric key K, challenger $\mathcal{C}$ with encryption scheme E_K.

Output: Whether the encryption scheme E_K is IND-CPA secure.

for polynomially many times **do**

$\mathcal{A}$ creates some message M_i.

$\mathcal{A}$ asks $\mathcal{C}$ for $C_i \leftarrow E_K(M_i)$.

$\mathcal{A}$ creates messages $M_0, M_1 \neq M_i$ and sends them to $\mathcal{C}$.

$\mathcal{C}$ fixes $b \in \{0,1\}$ and sends $E_K(M_b)$ to $\mathcal{A}$.

$b' \leftarrow A(M_0, M_1, E_K(M_b))$

$\mathcal{A}$ wins if $\Pr[b' = b] \geq \frac{1}{2} + o\left(2^{-|M_b|}\right)$, $\mathcal{C}$ wins otherwise.

In this game, we have a challenger and an adversary. The challenger first fixes $b \in \{0,1\}$. The challenger and adversary find a key using $K = G()$. Then while the adversary cannot guess b, he sends two messages M_0 and M_1 to the challenger. The challenger sends $C_b = E_K(M_b)$ to the adversary. Let b' be the adversary's eventual guess for b. Then if the encryption scheme passes IND-CPA, $\Pr[b = b'] = \frac{1}{2} + o\left(2^{-|M_0|}\right)$.

Example 4. Let $E_K(M) = 2M$. Certainly this is correct, with $D_K(C) = C/2$. But it does not pass IND-CPA.

Example 5. Let $E_K(M) =$ random number. Obviously this encryption scheme is not correct.

Example 6. Let $E_K(M) = K + M \pmod p$. One can show that $\Pr[b = b'] > \frac{1}{2}$.

For an IND-CPA correct scheme, we need the tools of the one-time-pad and the block cipher.

In the one-time-pad, the key is a bitstring $K \in \{0,1\}^n$, and the message is also a bitstring $M \in \{0,1\}^n$. Then $E_K(M) = K \oplus M$ and $D_K(C) = K \oplus C$. The one-time-pad is not in IND-CPA, but if we use it only once with a given key K, then it is perfectly secure (repeated keys allow you to XOR messages with each other to obtain K). In particular, given a key K, $b \in \{0,1\}$, and $C = K \oplus M_b$, then $\Pr[\text{Adversary decoding of } C = M] \leq o\left(2^{-|M_b|}\right)$, and $\Pr[b' = b] = \frac{1}{2}$, where b' is the adversary guess of b; this last equality is because all keys K are equally likely, so if $K_b = C \oplus M_b$, $\Pr[C = M_0 \oplus K_0] = \Pr[C = M_1 \oplus K_1]$.

Now, we cover block ciphers. The encryption function takes the form $E\colon \{0,1\}^{|K|} \times \{0,1\}^{|M|} \to \{0,1\}^{|C|}$, or for a specific key has $E_K\colon \{0,1\}^{|M|} \to \{0,1\}^{|C|}$. The function E_K must be invertible, so $|M| = |C|$. Then E_K must be a permutation. This scheme is secure if, for any codeword M chosen by the attacker, an attacker without K cannot distinguish the function E_K from a random permutation σ, given the outputs $E_K(M)$ and $\sigma(M)$. Then for each algorithm an attacker uses,

$$\Pr[\text{attacker wins}] \leq \frac{1}{2} + o(1)$$

where the "attacker winning" means that they can guess which permutation is the block cipher and which is the random permutation.

The attacker cannot recover M, since if so then the probability that they win is 1. Likewise, the attacker cannot recover any partial information, because they can then send another codeword with that information preserved and see which outputs preserve this information. For example, if the attacker observes that the least significant bit of M changes predictably under the block cypher, then they can send another message and measure the least significant bit.

One of the more recent implementations of a block cipher is AES, which is called "Advanced Encryption System". Another one is called "DES".

Block ciphers aren't IND-CPA secure. If the attacker sends two two-block messages, where one message M_0 has the same content in each half, and another message M_1 has different content, then we can tell which C_b we are given back based on if C_0 has the same first and second half (M_0) or not (M_1). Also, block ciphers are deterministic and thus cannot be IND-CPA.

A cipher is in ECB mode if the cipher splits up the message M into multiple blocks $M = \overline{M_1 \cdots M_k}$, then turns each of them into ciphertexts $C_1, \ldots, C_k$, then concatenates them to form $C = \overline{C_1 \cdots C_k}$. Clearly these are not in IND-CPA.

The CBC method splits $M = \overline{M_1 \cdots M_k}$. We need a key K and random initialization vector $C_0 \in \{0,1\}^{n/k}$. Then for each block i, do $C_i = E_K(M_i \oplus C_{i-1})$ and set $V = C_i$. Formally, the algorithm takes the form

Algorithm 2 CBC-Mode Block Cipher Encryption

Input: Message $M \in \{0,1\}^n$, Key $K \in \{0,1\}^r$.
Output: Ciphertext $C \in \{0,1\}^n$.
  $M_1, \ldots, M_k \leftarrow M$
  $C_0 \leftarrow$ random vector $\in \{0,1\}^{n/k}$
  for $i \in \{1, \ldots, k\}$ **do**
    $C_i \leftarrow E_K(M_i \oplus C_{i-1})$
  return $\overline{C_0 C_1 \cdots C_k}$

and given the key, we can do the obvious decryption process

Algorithm 3 CBC-Mode Block Cipher Decryption

Input: Ciphertext $C \in \{0,1\}^n$, Key $K \in \{0,1\}^r$.
Output: Message $M \in \{0,1\}^n$.
  $C_0, C_1, \ldots, C_k \leftarrow C$
  for $i \in \{1, \ldots, k\}$ **do**
    $M_i \leftarrow D_K(C_i) \oplus C_{i-1}$
  return $\overline{M_1 \cdots M_k}$

Another method is OFB mode, where each block is encrypted repeatedly and used as the key to a one-time-pad. In particular, the encryption process goes like this:

Algorithm 4 OFB-Mode Block Cipher Encryption/Decryption

Input: Message $M \in \{0,1\}^n$, Key $K \in \{0,1\}^r$.
Output: Ciphertext $C \in \{0,1\}^n$.

$M_1, \ldots, M_k \leftarrow M$
$Z_0 \leftarrow$ random vector $\in \{0,1\}^{n/k}$
for $i \in \{1, \ldots, k\}$ **do**
  $Z_i \leftarrow E_K(Z_{i-1})$
  $C_i \leftarrow E_K(M_i \oplus Z_i)$
return $\overline{Z_0 C_1 \cdots C_k}$

Due to the symmetry of the XOR operation, the encryption and decryption are the same.

Another mode is the Counter Mode, which is a parallizable CBC-type encryption.

Algorithm 5 CBC-Mode Block Cipher Encryption

Input: Message $M \in \{0,1\}^n$, Key $K \in \{0,1\}^r$.
Output: Ciphertext $C \in \{0,1\}^n$.

$M_1, \ldots, M_k \leftarrow M$
$C_0 \leftarrow$ random vector $\in \{0,1\}^{n/k}$
for $i \in \{1, \ldots, k\}$ **do**
  $Z_i \leftarrow E_K(C_0 + i)$
  $C_i \leftarrow E_K(M_i \oplus Z_i)$
return $\overline{C_0 C_1 \cdots C_k}$

The decryption is the same as the CBC-Mode.

For IND-CPA secure algorithms, any partial information does not help the attacker find the key or decrypt the whole message. This is a lot less powerful than the attacker's capabilities within the IND-CPA game, so this makes sense.

We now turn to pseudo-random number generators. Let $G\colon \{0,1\}^k \to \{0,1\}^n$ be a pseudo-random number generator. Then $G(K)$ generates n random bits, such that no statistical test can differentiate the distribution of bits from a random distribution.

This gives us a better version of the one-time pad, where the key K can be short; then $C = M \oplus G(K)$.

Asymmetric-Key Cryptography

Asymmetric-key cryptography wants to solve the problem of both the receiver and sender needing the same key. A generic scenario is that Alice has a key pair $K^A = \left(K^A_{\text{pub}}, K^A_{\text{priv}}\right)$. Everyone knows K^A_{pub}, but only Alice knows K^A_{priv}. Likewise Bob has $K^B = \left(K^B_{\text{pub}}, K^B_{\text{priv}}\right)$. A transmission from Alice to Bob has Alice encrypting with Bob's public key, then Bob decrypts with his own private key. There are three algorithms:

- $G() = (K_{\text{pub}}, K_{\text{priv}})$, returning a key pair $(K_{\text{pub}}, K_{\text{priv}})$.
- $E_{K_{\text{pub}}}(M) = C$, which returns the encrypted message C.
- $D_{K_{\text{priv}}}(C) = M$, which returns the original message M.

For correctness, we have that for all key pairs K that $D_{K_{\text{priv}}}\left(E_{K_{\text{pub}}}(M)\right) = M$.

Definition 7 (One-Way Function). A function f is **one-way** if computing $f(x)$ is easy (it takes polynomial time), but given any y finding any x such that $f(x) = y$ is hard (it takes super-polynomial time).

Example 8. $f(x) = x$ and $f(x) = 1$ are not one-way functions, but $f(x) = E_K(x)$ where E_K is specifically the block-cipher encryption scheme, is a one-way function, since E_K without knowledge of K is indistinguishable from a random permutation.

Definition 9 (Discrete Logarithm). Let $f(x) = g^x \pmod p$ where p is a large prime (2^k bits long) and $g \in [2, p-1]$. We assume that $f(x)$ is a one-way function. It's easy to compute (using binary exponentiation)

This discrete logarithm is used in the Diffie-Hellman key exchange.

Algorithm 6 Diffie-Hellman Key Exchange

Input: Alice and Bob, two practitioners of cryptography.
Output: A secret key K_S known to both Alice and Bob but nobody else.

Alice and Bob pick p, a large prime, and $g \in [2, p-1]$.
Alice picks at random $a \in [1, p-2]$; Bob picks at random $b \in [1, p-2]$.
Alice computes $A = g^a \pmod p$; Bob computes $B = g^b \pmod p$.
Alice sends A to Bob; Bob sends B to Alice.
Alice computes $s = B^a \pmod p = g^{ab} \pmod p$; Bob computes $s = A^b \pmod p = g^{ab} \pmod p$.
return s, which Alice and Bob can use for symmetric key cryptography.

Clearly since discrete logarithm is hard, Eve cannot compute $g^{ab} \pmod p$ so it's safe. Actually this requires that the attacker cannot compute $g^{ab} \pmod p$ from $g \pmod p$, $g^a \pmod p$, and $g^b \pmod p$.

This all works if Eve is only a passive eavedropper. If Eve actively changes the contents of the messages, she can send her own public key to Alice and Bob, spoofing the other person, in a **man-in-the-middle** attack. Then she can intercept all future communication between them. In particular, Eve will pick a private key m and forms a secret key s_{am} with Alice and a secret key s_{bm} with Bob. Then she will use s_{am} in symmetric-key cryptography to obtain whatever messages Alice is sending, and simultaneously use s_{bm} to send whatever messages she wants to spoof to Bob.

One way to beat the man-in-the-middle attack to have a separate, more secure channel, and Alice and Bob send transaction metadata simultaneously; if it doesn't match up, they know there is a man-in-the-middle attack.

The security game for asymmetric encryption is similar in spirit to IND-CPA. It's called Semantic Security. The challenger generates $G() \to (K_{\text{pub}}, K_{\text{priv}})$ and sends K_{pub} to the adversary. The adversary sends two messages M_0 and M_1 to the challenger. The challenger then chooses $b \in \{0, 1\}$ at random and sends $E_{K_{\text{pub}}}(M_b)$ to the adversary. Then the adversary guesses b', the bit of the message we sent. To win the game, the challenger needs $\Pr[b' = b] \leq \frac{1}{2} + \mathcal{O}\left(2^{-|M_0|}\right)$.

If the scheme is deterministic, the adversary can themselves carry out $E_{K_{\text{pub}}}(M_0)$ and $E_{K_{\text{pub}}}(M_1)$ and compare it with the challenger's returned $E_{K_{\text{pub}}}(M_b)$. So all schemes that pass Semantic Security are randomized.

This is all very abstract, so let's come up with a public-key encryption scheme.

Algorithm 7 The El Gamal cryptographic scheme.

function G
Generate at random a large (2048-bit) prime p
Generate at random $g \in [2, p-1]$
Generate at random $K_{\text{priv}} \in [2, p-2]$
$K_{\text{pub}} \leftarrow g^{K_{\text{priv}}} \pmod p$
return $(K_{\text{pub}}, K_{\text{priv}})$ ▷ Can assume that g and p are public.

function $E_{K_{\text{pub}}}(\ M\ )$
Assert $M \in [1, p-1]$
Generate at random $r \in [1, p-1]$
$C \leftarrow \textsc{Concatenate}\left(\underbrace{g^r \pmod p}_{C_1}, \underbrace{MK_{\text{pub}}^r \pmod p}_{C_2}\right)$
return C

function $D_{K_{\text{priv}}}(\ C_1, C_2\ )$
$M \leftarrow \frac{C_2}{C_1^{K_{\text{priv}}}} \pmod p$ ▷ $\frac{C_2}{C_1^{K_{\text{priv}}}} \pmod p = \frac{MK_{\text{pub}}^r \pmod p}{(g^r \pmod p)^{K_{\text{priv}}}} \pmod p = \frac{Mg^{K_{\text{priv}} r}}{g^{K_{\text{priv}} r}} \pmod p = M \pmod p$.
return M

One may note that the message cannot be 0. We can solve this via padding the message; in general, the encryption adds padding and the decryption removes the padding.

- The padding can be all 0s, but if the message ends with 0 then we lose since the decryption removes message bits.
- The padding can be all 0s preceded by a 1, but this works only for messages of size strictly less than the allowed size.

If we want to encrypt a very long message, we can encrypt a shared symmetric key K_{sym} and send it via the asymmetric encryption scheme. Then once both parties have K_{sym}, they can use symmetric key methods to send the long message, which is much more viable due to chaining methods.

Clearly one can break algorithm 7 by using discrete logarithm to extract M from C_1 and C_2, but the tightest condition for the security of algorithm 7 is that given p, g, $g^{K_{\text{priv}}}$, one cannot distinguish (C_1, C_2) from (C_1, R), where R is some random value.

Hashing

We wish to ensure authenticity of our message, to check that nobody has altered it. To this end we use hashing. Let a **hash function** $h\colon \{0,1\}^n \to \{0,1\}^L$ be a deterministic function. In reality, n can be whatever is required, but L is fixed.

Example 10. In SHA256, $L = 256$, for example.

The term $h(x)$ is the **hash** of x, the **digest** of x, or the **fingerprint** of x.

Ideally, hash functions are

- One-way (pre-image resistant, or hard to invert): if x is chosen uniformly at random from $\{0,1\}^n$, and if $y = h(x)$, then across all adversaries, the best guess $A(y)$ of x obeys $\Pr[x = A(y)] = o\left(2^{-L}\right)$.
- Collision resistant: it is computationally infeasible (takes super-polynomial time) to find (x, x') such that $x \neq x'$ and $h(x) = h(x')$. Currently, SHA256 is assumed to be collision-resistant.
- Correctness: h should be deterministic.
- Efficiency: computing $h(x)$ for any x should take polynomial time.
- Security:

Note that by Pigeonhole Principle, there are many collisions; they are just hard to find.

Here is how hashing works. Say Alice is trying to download something from Bob, who provides the file x and, through a different and secure channel, a hash $h(x) = h_b$. Alice downloads the file x' and computes the hash $h(x') = h_a$. If $h_a = h_b$, then either (x, x') is a collision, or $x = x'$; we assume collision resistance, so we assume $x = x'$ and the download is successful. On the other hand, if $h_a \neq h_b$ then the download is in some way compromised.

Cryptography algorithms are not meant to provide integrity and authenticity; they are meant only to preserve message confidentiality.

Symmetric-key cryptography works well with message authentication codes (MACs; a common implementation is AES-EMAC) to provide integrity and authenticity. When Alice sends $C = E_K(M)$ to Bob, Alice also sends $\text{MAC}_K(M) = T$ to Bob. On Bob's end, he decrypts $M = D_K(C)$ and computes $T' = \text{MAC}_K(M)$, verifying that $T = T'$.

This scheme is correct because it is deterministic. Computing $\text{MAC}_K(M)$ should take polynomial time, so the scheme is efficient. We also want the function MAC_K to not be forge-able, which means that the scheme is EU-CPA (existentially unforgeable against chosen plaintext attack). Correspondingly, there exists a game for it.

Algorithm 8 The EU-CPA game.

Input: Adversary $\mathcal{A}$ with guessing function A, randomly generated symmetric key K, challenger $\mathcal{C}$ with hash scheme MAC_K.
Output: Whether the scheme MAC_K is EU-CPA-secure.
  for polynomially many times **do**
    $\mathcal{A}$ asks $\mathcal{C}$ for $T_i \leftarrow \text{MAC}_K(M_i)$
  $\mathcal{A}$ creates a message M.
  $\mathcal{A}$ wins if $\Pr[A(M) = \text{MAC}_K(M)] \geq o\left(2^{-|T_i|}\right)$ across all K selected uniformly at random.

Algorithm 9 The AES-MAC scheme.

Input: Key K selected uniformly at random from $\{0,1\}^{256}$, hash scheme AES_K, message M.
Output: $T \leftarrow \text{AES}_K(M)$
  $M \leftarrow \overline{M_1 \cdots M_L}$ (128 bit blocks).
  $K \leftarrow \overline{K_1 K_2}$
  $S_0 \leftarrow 0$
  for $i \in [1, L]$ **do**
    $S_i \leftarrow \text{AES}_{K_1}(M_i \oplus S_{i-1})$
  return $T \leftarrow \text{AES}_{K_2}(S_L)$

The AES-EMAC is not collision-resistant when the adversary has the key K. We now introduce the AES-HMAC function:

$$\text{HMAC}_K(M) = h(K \oplus \phi_1 || h((K \oplus \phi_2)||M))$$

where h is some one-way collision-resistant hash function, $\phi_1 = \texttt{0x5c5c...5c}$, and $\phi_2 = \texttt{0x3636...36}$.

We want to show that HMAC is collision-resistant. If

$$\begin{aligned}
\text{HMAC}_K(M_1) &= \text{HMAC}_K(M_2) \\
K \oplus \phi_1 || h((K \oplus \phi_2)||M_1) &= K \oplus \phi_1 || h((K \oplus \phi_2)||M_2) \\
K \oplus \phi_2 || M_1 &= K \oplus \phi_2 || M_2 \\
M_1 &= M_2
\end{aligned}$$

This is therefore a nice construction for both a hash and MAC.

We now introduce digital signatures, which are the public-key versions of MAC. Alice produces a signature with $S \leftarrow \text{Sign}_{K^A_{\text{priv}}}(M)$; Bob recieves $C \leftarrow E_{K^B_{\text{pub}}}(M)$ and S, then decrypts $M' \leftarrow D_{K^B_{\text{priv}}}(C)$ and does $\text{Verify}_{K^A_{\text{pub}}}(M') \in \{\texttt{TRUE}, \texttt{FALSE}\}$ – whether the message integrity is preserved or not. The security game is still EU-CPA and tests whether the attacker can recreate a digital signature.

RSA is an example of a digital signing scheme. The key generation picks two random primes p and q that are both $p \equiv q \equiv 2 \pmod 3$. The public key is $K_{\text{pub}} = pq$. Let $\phi(n) = |\{x \geq 0 \colon \gcd x, n = 1\}|$. In particular, $\phi(K_{\text{pub}}) = (p-1)(q-1) = \text{ord}\left(\mathbb{Z}_{K_{\text{pub}}}\right)$. Note that $a^{\phi(K_{\text{pub}})} \equiv 1 \pmod{K_{\text{pub}}}$ for all a. Then K_{priv} is such that $3K_{\text{priv}} \equiv 1 \pmod{\phi(K_{\text{pub}})}$. Then $\text{Sign}_{K_{\text{priv}}}(m) = h(M)^{K_{\text{priv}}} \pmod{K_{\text{pub}}}$ and $\text{Verify}_{K_{\text{pub}}}(M, S) = \mathbb{1}(S^3 \pmod{K_{\text{pub}}} = h(m) \pmod{K_{\text{pub}}})$.

We wish to prove correctness of this scheme. We have that

$$\begin{aligned}
\left(h(M)^{K_{\text{priv}}}\right)^3 \pmod{K_{\text{pub}}} &= h(M)^{3K_{\text{priv}}} \pmod{K_{\text{pub}}} \\
&= h(M)^{r\phi(K_{\text{pub}})+1} \pmod{K_{\text{pub}}} \\
&= 1^r h(M) \pmod{K_{\text{pub}}} \\
&= h(M) \pmod{K_{\text{pub}}}
\end{aligned}$$

as desired. A necessary assumption for security is thus that no adversary can factor large numbers.

Key Management

Normally, Alice might ask Bob for his public key K^B_{pub} in order to send a message to him. But, if an attacker Eve intercepts this communication and is able to themselves send messages, they might be able to send their public key K^E_{pub} to Alice and recieve Bob's public key K^B_{pub}. Then they can receive encrypted messages from Alice using $E_{K^B_{\text{pub}}}$ and send encrypted messages to Bob using $E_{K^A_{\text{pub}}}$, spoofing one person to the other. We want to protect against this.

One such way is the trusted directory. Alice can query the trusted directory D for Bob's public key K^B_{pub}, assuming that the trusted directory has the correct key. However, the adversary Eve can perform a man-in-the-middle attack in the transaction between Alice and the trusted directory, and we haven't gotten anywhere. Instead, we can use the digital signature approach. Assume that Alice has K^D_{pub} previously known, and queries D for K^D_{pub}. Then D can reply with K^B_{pub} and $\text{Sign}_{K^D_{\text{priv}}}\left(K^B_{\text{pub}}\right)$. However, this can still be beaten by a man-in-the-middle attack. The way Alice wins is to embed a nonce in her request,

and checks the signature from D to ensure that it contains the nonce and Bob's name. The drawbacks of this approach are scalability (a single directory has to store and serve all public keys), the central point of attack and trust that the directory represents, the directory has to be always available, and so on.

Another approach is to use a digital certificate. A certificate authority C provides an association between a name and a public key. The certificate forms this authority. A sample certificate takes the form $\text{cert}^B = \text{Sign}_{K^C_{\text{priv}}}\Big(\text{Bob's public key is } K^B_{\text{pub}} || \text{Bob's key expires on date } X\Big)$. Anyone can serve K^B_{pub} and cert^B. Alice checks cert^B verifies with K^C_{pub}, and is not expired, and the key actually matches K^B_{pub}. Alice therefore no longer contacts D to fetch K^B_{pub}.

We now deal with hierarchical key management. Alice$A$ requests a public key from a trusted site B, which has K^B_{pub} and K^B_{priv}. B uses a certificate authority C) to generates a signature $\text{cert}^B = \text{Sign}_{K^C_{\text{priv}}}\Big(B \text{ has } K^B_{\text{pub}}; \text{expiry date}\Big)$. Then B sends $\Big(K^B_{\text{pub}}, \text{cert}^B\Big)$ to A; A has hardcoded K^C_{pub}, so she can verify cert^B and the expiry date.

A certificate hierarchy is when intermediates want to act as certificate authorities. Each "authority" only certifies the level underneath. Every transaction requires all certificates from each intermediate on the way to the ultimate authority.

Passwords are key sources with reduced amount of entropy, but are easier for the user to use and memorize. There are several attacks and defenses around password security:

- A network eavesdropper will grab the password in transit, but one can encrypt the password client side.
- Client side malware can grab the password on client side; this is hard to defend against, but two-factor authentication helps.
- Online guessing attacks (brute-force, or dictionary attacks) fail against rate-limiting, password requirements, and CAPTCHAs.
- The server being compromised doesn't matter too much if all the passwords are salted and hashed.

5 Networking

Let's start with local area networks (LANs). There are two kinds of LANs:

- Point-to-point: two computers are corrected via a wire.
- Shared: multiple computers access the same wire and can share the wire for communication. Each packet has a source, destination, and message.

A router connects two LANs to create a WAN (wide area network). The router is the destination of cross-LAN packets, and it forwards the packets within the respective LANs.

A protocol is an agreement on how to communicate. It includes a syntax (how a communication is specified and structured, such as format, order of sending and receiving, etc.) and semantics (what a communication means, such as actions taken when transmitting, receiving, timer expires, etc.).

The original Internet design prescribes that the routers have no knowledge of ongoing connections going through them. Each packet is self-contained. The routers look at the destination address to forward.

Internet design is partitioned into layers. Each layer relies on services provide by the next layer below and provides services to the layers above it. A standard layering (OSI model) is

- Application
- Transport
- (Inter)Networking (IP Protocol)
- Link (Ethernet)
- Physical (Electronics)

Modern networks break communications up into packets. For our purposes, packets contain a variable amount of data up to a maximum specified by the particular network. The sending computer breaks up the message and the receiving computer puts it back together. So the software doesn't actually see the packets per-se. The network itself is packet-switched, which means that it sends each packet towards its next destination.

Packets are received correctly or not at all, if random errors occur. Packets have a `checksum`. There are no guarantees if the adversary modifies packets (there are no cryptographic MACs). Packets may be unreliable and "dropped"; it's up to higher-level protocols to make the collection reliable.

A single packet has the view:

- Link Layer Header
- (Inter) Network Layer Header (IP)
- Transport Layer Header
- Application Data: the structure depends on the application.

The physical layer encodes bits to send them over a single physical link. The network layer protocol connects multiple sub-nets. The transport layer provides end-to-end communication between processes – the different services provided are TCP (reliable byte streams) and UDP (unreliable datagrams). The application layer communicates whatever is desired, and can use whatever transport is convenient.

An IP Packet Header contains a pair of IP addresses. Each source has an address, which is a unique identifier and locator for the sending host. The recipient can decide whether to accept packets. It enables the recipient to send a reply back to the source. Each destination has an address, which is a locator for the intermediate routers.

Packet delivery is a "best effort" service. Routers inspect the destination address and locate the "next hop" in the forwarding table. Packets may be lost, corrupted, and/or delivered out of order.

An alternative to "best effort" is to use TCP, which is connection oriented, and multiple end hosts can have multiple long lived communications. This provides reliable, in-order byte-delivery.

With IP we think in terms of packets. With TCP we think in terms of connections. A connection has a bytestream; or actually, a pair of bytestreams, one going in each direction.

A TCP header has source ports, destination ports, and more metadata. TCP assigns each port on your machine to a different connection. A header also has a sequence number, which is the byte offset of the data carried in the packet in the bytestream. The acknowledgement segment gives the sequence number just beyond the highest sequence number recieved in order; if the sender sends n bytestream bytes starting at sequence number s then the acknowledgement foor it will be $s + n$. There are also some flags, which are instances of a bitset.

There's a three-way handshake to establish connections:

- Host A sends a `SYN` (open; "synchronize sequence numbers") to host B

- Host B sends a `SYN` acknowledgement (`SYN` + `ACK`)
- Host A sends an `ACK` to acknowledge the `SYN` + `ACK`

Each host tells its initial sequence number (ISN) to the other host.

Host names are mnemonic names appreciated by humans; IP addresses are numerical addresses appreciated by routers, are fixed length binary numbers, and are hierarchical, related to the host location. The DNS assigns host names to IP addresses.

Let's talk about link-layer attacks. Eavesdroping, injection of spoofed packets, and intercepting packets are all possible.

For subnets using broadcast technologies, the attacker can eavesdrop. Each attached system's NIC can capture any communication on the subnet.

An attacker can inject spoofed packets and lie about the source address. It's particularly powerful when combined with eavesdropping, because the attacker can understand the exact state of the victim's communication (i.e. sequence number) and craft their spoofed traffic to match it. This is much easier if the router of the attacker is in the communication path.

If A sends a TCP packet with `RST` flag to B and the sequence number fits, the connection is terminated unilaterally and immediately. The attacker can inject `RST` packets and block connections.

If the attacker knows the ports and sequence number (i.e. if they are at least as knowledgeable as an on-path attacker), the attacker can inject data into any TCP connection, with the receiver being none the wiser. This is called TCP **connection hijacking** or **session hijacking**. It's not really possible for off-path attackers to inject into the TCP connection unless they can guess the sequence numbers.

To guess the sequence numbers, it suffices for the attacker find the initial sequence number used by the target; this initial sequence number is a simple function of the system clock. This is predictable because the attacker can create a connection with the target and find the initial sequence number of that transition, then synchronize the clocks, and thereby find a range of values for the initial sequence number of the first transaction, which can then be brute-forced so that the attacker can do a session-hijacking.

One possible solution is to compute an initial sequence number randomly, or compute it using a cryptographic key-based procedure. This has rendered such attacks moot.

If a new host doesn't have an IP address yet, and has no idea who to ask for an IP address, then it broadcasts a server-discovery packet; the DHCP server replies with the 'offer' (an IP address, DNS server for looking up hostnames, gateway router that the client uses as the first hop for all of its traffic, and lease time).

A possible attack is to run a malicious DHCP server and edit the gateway router or DNS server. Solving this problem is hard because they don't have a trust anchor to obtain a DHCP offer from.

A domain name is a human-friendly name to identify a server or service. They are arranged hierarchically. The top-level domain (TLD) is something like `.com`, and subdomains are like `google.com`, and so on.

To resolve a domain name, a resolver queries a distributed hierarchy of DNS servers, also called name servers. At the top level are the root name servers, which resolve TLDs. Users store the authoritative name server for each TLD (the trusted server for the TLD).

DNS usually uses UDP for its transport protocol. Frequently, both clients and servers use port 53. The UDP packet has an IP header, source port, destination port, checksum, length, and payload. The DNS-specific

payload has identification, flags, number of queries, number of answers, number of authorities (name servers responsible for answer) queried, and number of additional sources (information client is likely to look up anyways) queried, then that information stacked in that order. Each resource record has a time to live (in seconds) for caching.

The issue is that nothing is authenticated. If there is a man-in-the-middle attack or the queried endpoint server is untrustworthy, then the adversary can change the authority information and thereby redirect all traffic to an (adversarial) DNS server of their own choosing. The fix here is for the client to not accept additional records unless they're for the domain we're looking up. There's no extra risk in accepting these since the server could return them to us in an answer anyways. However, a compromised DNS server can still return whatever the adversary wants.

If the attacker cannot eavesdrop on the attack, then the only thing that can be done is blind spoofing, where the attacker responds to queries before the legitimate server replies. The adversary can use some probabilistic methods, or trace the packet routing, from the client, and figure out the source and destination of the query. Finding the identification is a bit harder; for example, the attacker can coerce the user to visit a malicious webpage and then figure out the identification number from that particular DNS transaction. The attacker can get around the caching of legitimate replies by spoofing several records, each with a different identifier.

The central problem is that the identification field has low entropy; using random destination ports makes it much harder to guess the correct identification number.

DNSSEC

We now cover the standard security considerations for DNS, or DNSSEC. It aims to ensure the integrity of the DNS lookup results.

One idea is to do DNS lookups over TLS (SSL), basically use cryptography to verify integrity of DNS connections. TLS is a protocol for building an encrypted connection, using public-key exchange to exchange a session key, then using encryption and a message authentication code on all data sent over the connection.

The downside of this idea is that the performance is pretty poor, makes it infeasible to use caching, and the DNS resolver is in the trusted computing base, but this is not always the case.

A modification of this idea is to make DNS results like certificates, i.e. verifiable signatures that guarantees who generated a piece of data; signing happens off-line. In particular, when we look up some site, we ensure the returned result is correct by having the site name server sign the certificate. The signature should contain the domain name, IP address, and cache time. We know the public key of the site is correct by having the name server one level above the site give the certificate on it. This chains until the root name server; the root name server public key is hardcoded into the resolver. Then the chain of certificates is verified using the corresponding public keys.

The issue with this idea is that signing is slow, so denial of service attacks become more powerful.

To sign the no-record responses offline, we can internally store an ordered list of signatures, and sign only the domains that exist; therefore if the signature is not in the cached list, the domain does not exist.

DoS

Today we discuss denial-of-service attacks. We can deny service via a program flaw or resource exhaustion. Either we can consume resources using an intentional leak in a memory-unsafe software, or spam billions of packets at a network. One example of this is amplification, where the reply from the host is much bigger than the request, and therefore requests take much less effort to generate than replies. Filtering malicious traffic sounds easy, but those filters can be evaded, and the hosts of the suspicious network traffic can be spoofed.

TCP traffic can be used as a method of amplification DoS. The attacker can initiate many TCP SYN packets; the server needs to create a state (buffers, timers, and counters) associated with the connection here, which is a lot bigger of a burden. This method can overload the server's memory capacity. A counter defense against this is to encode the connection state within the SYN-ACK sequence number, and when the malicious ACK responding to the SYN-ACK packet arrives we check that the sequence numbers check out, and then (and only then) create the internal state. The takeaway here is that rather than holding state, we encode it so that it's returned as needed.

Firewalls

The motivation for firewalls is to harden a system against external attack. The more network services the machines in the system run, the greater the risk of attack. We can mitigate this risk by blocking (via the network) outsiders from having unwanted access to network services. To this end we interpose a firewall the traffic to/from the outside of the system must traverse. The chokepoint can cover up to thousands of hosts.

The firewall enforces an acccess control policy. Conventional implementations distinguish betweeen inbound and outbound connections. Inbound connections may be attempts by internal users to connect to services on internal machines. Outbound connections may be attempts by internal users to connect to external services, services for which the system is not responsible for. The access policy is simple: we permit inside users to connect to any service, but external users are restricted in that connections are only to services meant to be externally visible, and all other connection attempts are denied. The default deny policy means that only a few inbound connections are allowed, and more are added when users complain.

A stateless packet filter inspects each packet for certain filtering rules to determine whether to pass or block it (with no history). The simplest policy is to deny all inbound connections, allow all outbound connections, and allow all inbound packets that are a reply. However, we can blind spoof this and send inbound packets that look like they're replies but actually aren't. In the TCP protocol, we can instead allow all outbound TCP packets and allow all inbound TCP packets with ACK flag set; we can't handle UDP connections.

A stateful packet filter is a router that checks each packet against security rules and deciedes how to forward or drop it. The firewall keeps track of all connections (both inbound and outbound). Each rule specifies which connections are allowed or denied. A packet is forwarded if it is part of an allowed connection. A connection is a source IP, a destination IP, and a destination port.

We can also have application-level firewalls, that act as proxies; these eliminate the risk of the stateful packet filter interpreting packets different from the end host.

Often, it is necessary to provide secure remote access to a network protected by a firewall. We create a secure channel (a Virtual Private Network) to tunnel traffic from outside the host/network to inside it. This may allow bypassing the firewall, reducing the firewall effectiveness.

A disadvantage of firewalls is that a single breach of the perimeter by an attacker means that you cannot

make any assertions about subsequent internal state i.e. a firewall is a single point of failure for the system.

Intrusion Detection

One way to detect network intrusion is to examine the network traffic. The benefit of this method is that there's no need to touch or trust the end systems, we can cover many systems with a single monitor, and management of many systems is centralized. This is done via a NIDS (network intrusion detection system), which has a table of all active connections and maintains a state for each. When it sees a new packet not associated with any known connection, it creates a new connection.

Evasion attacks hapepn when we parse a packet at both the firewall monitor and the end system. They exploit semantic inconsistency and ambiguity. Some evasions reflect incomplete analysis; in principle, we can deal with these with implementation care. Some are due to imperfect observability, e.g. two copies of the same packet, which are actually different and with different TTLs.

Another approach is to instrument the web server; this is a host-based IDS (sometimes called HIDS). We scan for arguments sent to back-end programs.

A third approach is to analyze log files whenever convenient (i.e. nightly). This has some large amount of latency and is naturally exploitable.

A fourth and last approach is to monitor system calls, which reduces the amount of complexity of the detection system but sometimes catches false positives. One interesting idea for detection is to look for activity that matches the structure of a known attack. Alternatively, we can match on known problems with the infrastructure, which are quite common in most applications. We can also match on anomalies on security systems, but if we aren't careful about what constitutes an "anomaly", this method is very ineffective, since the attack can look very similar to normal behavior. This has an alteration where we specify that only "normal" behavior is allowed, but this is expensive in terms of effort to classify what's allowed for each purpose. Finally, we can look for evidence of compromise instead of attacks, but this obviously gives up the chance of preventative measures.

Evasions arise from uncertainty or incompleteness because the detector must infer the behavior of processes that it can't directly observe. One general strategy is to impose content restrictions on interactions (i.e. remove escapes or injections). Another strategy is to analyze all possible interpretations rather than assuming one, of any content. Another strategy is to flag potential evasions, so the presence of an ambiguity is at least noted. Another strategy is to fix the basic observation problem, e.g. monitor directly at the end systems.

Modern day antiviruses do all of this and more. Firewalls (NIDS) and antiviruses (HIDS) are used in concert to protect systems.

Getting around firewalls requires the existence of secure end-to-end channels. The basic idea is that the client picks some symmetric keys for encryption and authentication, the client sends them to the server (encrypted), both sides send message authentication codes, and then the client and server use these keys to encrypt and authenticate all subsequent messages.

6 Secure Channels

We've talked about TLS, or transport layer security. It's a secure channel for applications that use TCP. Secure, in that it ensures encryption/confidentiality, integrity, and authentication (of the server, but crucially

not of the client).

RSA is an important building block of TLS. RSA was designed as a digital signature scheme, but it can also be used as a public key encryption algorithm. The encryption algorithm is similar to the verify algorithm, and the decryption is similar to the signing algorithm. We need to encrypt the message using a special padding, but that's a small difference.

Here's how TLS works.

- Browser (client) picks some symmetric keys for encryption + authentication.
- Client sends them to server, encrypted using RSA public-key encryption
- Both sides send MACs
- Now they use these keys to encrypt and authenticate all subsequent messages, using symmetric-key crypto

Let's look at an example:

- Browser (client) connects via TCP to Amazon's HTTPS server
- Client picks 256-bit random number R_B, sends over list of crypto protocols it supports
- Server picks 256-bit random number R_S, selects protocols to use for this session
- Server sends over its certificate
- Client now validates certificate
- For RSA, browser constructs "Premaster Secret" S_P
- Browser sends S_P encrypted using Amazon's public RSA key K^A
- Using S_P, R_B, and R_S, browser and server derive symmetric cipher keys (C_B, C_S) and MAC integrity keys (I_B, I_S) – one pair to use in each direction.
- Browser and server exchange MACs computed over entire dialog so far
- If good MAC, browser displays green lock symbol
- All subsequent communication encrypted w/ symmetric cipher (e.g., AES128) cipher keys, MACs (sequence numbers thwart replay attacks)

An alternative is to use Diffie-Hellman key exchange.

- Browser (client) connects via TCP to Amazon's HTTPS server
- Client picks 256-bit random number R_B, sends over list of crypto protocols it supports
- Server picks 256-bit random number R_S, selects protocols to use for this session
- Server sends over its certificate
- Client now validates certificate
- For DH, server generates random a, sends public parameters and $g^a \pmod p$, signed with the server's private key.
- Browser verifies the seignature using the public key from the certificate
- Browser generates random b, computes $S_P = g^{ab} \pmod p$, and sends its public key to server
- Server also computes $S_P = g^{ba} \pmod p$.
- Using S_P, R_B, and R_S, browser and server derive symmetric cipher keys (C_B, C_S) and MAC integrity keys (I_B, I_S) – one pair to use in each direction.
- Browser and server exchange MACs computed over entire dialog so far
- If good MAC, browser displays green lock symbol
- All subsequent communication encrypted w/ symmetric cipher (e.g., AES128) cipher keys, MACs (sequence numbers thwart replay attacks)

We consider a principle named **forward secrecy**. If the attacker steals the long term secret key of the server, they should not be able to read past conversations, or compromise any past secret keys. TLS with RSA does not have forward secrecy; TLS with DH does.

7 Web Security

In the framework of the current internet, the client sends a HTTP request to the server, and the server sends an HTTP response. Requests that are of type GET have no side requests; type POSTs have side requests, sometimes.

Web pages have three components for the purposes of this class: HTML, CSS, and JavaScript. Most exploits are done through JavaScript, since it's significally more powerful (Turing-complete) than the other two components.

When the browser wants to access a webpage, it first reads the HTML through a parser. The parser creates a DOM (Document-Object-Model) tree. The CSS then is interpreted and fed through the parser, which augments the DOM tree. Finally, the JavaScript is read and modifies the DOM tree in arbitrary ways. Finally, the browser takes the DOM tree and paints it to the screen.

Let's cover the DOM more. The DOM is a cross-platform model for representing and interacting with objects in HTML. It's a tree that mirrors the HTML tag tree. JavaScript can alter the DOM in almost arbitrary ways.

Another security-significant feature of HTML is to embed a page inside another page using a frame. This helps with composing a webpage, via modularity and delegation of duties to websites. On the other hand, the outer page can only specify sizing and placement of the frame in the outer page; the inner page cannot change the contents of the outer page.

In web security, we have slightly different implementations of the regular security goals.

- Integrity: malicious web sites should not be able to tamper with integrity of my computer or my information on other web sites
- Confidentiality: malicious web sites should not be able to learn confidential information from my computer or other web sites
- Privacy: malicious web sites should not be able to spy on me or my activities online
- Availability: attackers should not be able to take regular web sites online

We do not want malicious websites to trash the machines. The defense is that JavaScript is sandboxed; we want to avoid security bugs in the browser code, privilege separation, automatic security updates, and so on.

We do not want malicious websites to be able to spy on or tamper with our information or interactions with other websites. The defense is the "same-origin policy", where each site in the browser is isolated from all others, and two sites can share information/resources if and only if they have the same origin (protocol, hostname, and port). The origin of a page is derived from teh URL it was loaded from; JavaScript runs with the origin of the page that loaded it. Cross-origin communication is allowed through a narrow API `postMessage`; the reciving origin decides to accept the message based on the seneding origin (whose correctness is enforced by the browser).

We want data stored on a web server to be protected from unauthorized access. The defense is general server-side security. Some web attacks to compromise the webserver are:

- SQL injection
- XSS – Cross-Site Scripting
- CSRF – Cross-Site Request Forgery

SQL Injection

The general code injection attack format is

- Attacker provides bad input to the web server
- The web server does not check input format
- Enables attacker to execute arbitrary code on the server

It's similar in scope to the buffer overflow attack.

There are functions in PHP and similar language that will evaluate any text as if it were code. The webserver runs a GET/POST request on some expression that the user inputs, then evaluates the expression at some point using one of those functions. Attacker can input a multi-line expression that correctly ends the benign expression and then i.e. deletes everything.

Let's now discuss SQL injection. The browser interacts with the (stateless) web server, and the web server interacts with the (stateful) database server. As a result, the browser can interact with the database server by proxy, and sometimes manipulate queries sent to the database server.

Databases are structured collections of data. They store tuples of related values, organized in tables. They are widely used by web services to store server and user information. The database runs as a separate process.

SQL is a widely used database query language. To fetch a set of rows we do `SELECT column FROM table WHERE condition` returns the values of the given column in the specified table, for all records where `condition` is true. To add data to the table (or modify) we can do `INSERT INTO table VALUES (tuple)`. We can drop entire tables with `DROP TABLE table`. We can concatenate multiple commands using semicolons.

The basic picture is that the attacker POSTs a malicious form with bad inputs. The victim web server gives an unintended SQL query to the SQL database. The SQL database returns secret data, or does unintended actions.

A lot of SQL injections are based around escaping the string that the form input is fed into as part of the SQL query, then adding in another SQL statement. To beat SQL injections, the backend programmer can go through each input and sanitize it, escaping all the quotes.

Cross-Site Scripting

The basic idea is that the attacker injects malicious JavaScript into a dynamic website the user visits. The important part is that scripts are executed by the browser. Scripts can alter page contents, track events (mouse clicks, motion, and keystrokes), and issue web requests and read replies.

Given all this power, browsers need to make sure JavaScript scripts don't abuse it. Recall that the browser associates the web page elements (text, layout, events) with a given origin.

Two main types of XSS are:

- Stored XSS: attacker leaves JavaScript lying around on the benign web service for victim to load

- Reflected XSS: attacker sends malicious script to web server, who reflects it at the user

In stored (persistent) XSS, the attacker manages to store a malicious script at the web server. The web server stores this script in a database. Then when the user logs into the web server, the user is served the script and executes it as though the server intentionally gave this script. This can allow the attacker to send HTTP requests, including those that save user information. This happens when the server does not sanity-check what is stored. People can embed JavaScript in many things, including arbitrary text, and images.

In reflected XSS, the attacker gets the victim user to visit a URL for the website that itself embeds a malicious JavaScript, usually on a vulnerable web service that will include parts of URLs it receives in the web page output it generates. The server echoes the JavaScript back to the victim user in its response; since this script is generated by the web page, it fulfills the same origin policy. To combat this, the web page should sanitize the URL inputs.

To defend against this, the web server must perform input validation (check that inputs are of an expected form) and output escaping (escaping dynamic data before inserting it into HTML). Another way to product is to have the web server supply a whitelist of the scripts that are allowed to appear on a page.

Session Management

Cookies are a way of maintaining state in the browser. They're stored in the browser on a site-by-site basis. The first time we do an HTTP GET request to a server, it responds with the regular response, and also a cookie, if it obtains via JavaScript that the client has no cookies for that site. The browser maintains a cookie jar with all the unexpired cookies it contains so far.

When the browser connects to the same server later, it automatically attaches the cookies in scope: the header containing the name and value, which the server can use to connect related requests; the server responds with a path and domain which define which paths and domains the cookie can be sent to. The cookie can have a secure flag, which indicates that it is to be sent over HTTPS only. The client can delete a cookie by setting the expiration field to a date in the past. There is also a flag (HTTPOnly) that signals that JavaScript cannot access the cookie; however, this does not prevent XSS.

The cookie policy has two parts:

- What sources a URL-hostname web server is allowed to set on a cookie
- When does the browser send a cookie to a URL

When the browser is deciding whether to accept a cookie, the browser checks if the web server may set the cookie by checking the scope, and if not, it will not accept the cookie. The domain field of the scope is any domain-suffix of URL-hostname of the webserver, except the TLD. The path field of the scope can be set to anything.

When the browser is trying to send a cookie, the goal is that the server only sees cookies in its scope, in other words, the cookie domain is a domain-suffix of the URL-domain, and the cookie path is a prefix of the URL-path, and if the cookie is HTTPS only then the connection between the client and server is secure.

Recall that when the browser sends a cookie, the browser will automatically attach all the cookies in scope. The scope is all domains where the cookie-domain is a domain-suffix of the domain, and all paths where t he cookie-path is a prefix of the path.

A session is a sequence of requests and responses from one browser to one server. Sessions can be long or short.

The earliest form of session management is HTTP authentication. There is one username and password for a group of users. The HTTP request is `GET /index.html`; the response contains `WWW-Authenticate: Basic realm="Password Required"`; the browser sends the hashed password on all subsequent HTTP requests. This is bad, because it's vulnerable to replay attacks, sniffing password, and usability concerns.

Now we use session tokens. On the first time a user accesses the website, the site serves an anonymous session token, which can be stored in a cookie. The browser attaches the cookie to each GET request. Upon a log-in, the website checks credentials on the username, and then elevates the session token to a logged-in session token, which is again embedded in a cookie. At POST requests, the server validates the logged-in token.

Embedding session tokens in browser cookie opens us up to CSRF since we send every applicable cookie with every request. Embedding session tokens into URL implies that tokens can leak via HTTP Referrer header. Embedding session tokens in a hidden form field only makes it possible to have short sessions.

CSRF

HTML forms allow a user to provide some data which gets sent with an HTTP POST request to a server. The browser always appends the relevant cookies to the POST request. The server assigns a random session token and places it in the cookie. The server keeps a mapping form username to sesion token, so when it sees the sesion token it knows which user it is. When the user logs out, the server clears the session token.

The attack first gets the user to establish a session to a sensitive web server where they have some desired privilege; they get a cookie for this website with the session token. Then the attacker visits the malicious server, and receives a malicious page, which upon loading can send a forged request to the sensitive web server (with the elevated cookie) via creating and sending a hidden form.

To combat this, the server stores a state that binds a random CSRF token to the user's session ID. Then the CSRF token is embedded in every form. On every request the server validates that the supplied CSRF toekn is associated with the user's session ID.

The last class of phishing attacks is UI attacks, where the UI is misleading. We can hide the target with our own applet, or partially overlay the window, so that the victim only knows the information we give them or allow them to see. This is a click-jacking attack.

In general, click-jacking is when an attack application/script compromises the context integrity of another application's user interface when the user acts on it. Two types of integrity are visual integrity and temporal integrity.

The defense against click-jacking attack is to have frame-busting code. This stops the frame vector of click-jacking. A common frame-busting method is that inner frame can take over the whole outer page.