50.020 Security Mid-term Recap

Basic terminology: Properties

"Classic" C,I,A properties:

- C onfidentiality
  - Attacker cannot obtain secret data of victim
- I ntegrity
  - Attacker cannot change data of victim undetected
- A vailability
  - Attacker cannot stop services provided by victim (Denial of Service/DoS)

Additional properties

- Non-repudiation
  - Attacker cannot deny having taken certain actions
- Privacy
  - An attacker cannot learn *private* information of victim
- Authenticity, . . .

Measuring security in practice

- Metrics such as number of bugs found in software
- Attack surface metrics: *how many entry points?*
- Practical time-to-compromise for experts
- In general: estimates based on complexity and cost

Effort/time estimates based on brute force key exploration:

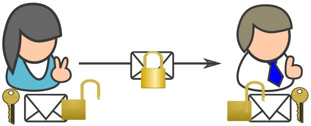
Key length	Attempts	Time to brute force attack
32	2^{32}	Realtime
64	2^{64}	Few days or less
128	2^{128}	Decades
256	2^{256}	Long term secure

Numbers for symmetric keys. See also:
`https://www.keylength.com/en/3/`

Basic terminology: Alice, Bob, and Eve

- Who are they?
  - Commonly used in security research to explain protocol interactions
  - Names sometimes change (Mallory, Charly, etc)
  - Just a convenient way to identify parties (e.g., servers, users)
  - Alice usually initiates communication
- Part of our fundamental attacker and system model (more later)

Basic terminology: Cryptography



- Alice wants to send a secret *message m* to Bob
- The original message m is the *plaintext*
- Alice has shared *key* k and symmetric encryption function E(m,k)
- Alice encrypts the plaintext to obtain a *ciphertext* c=E(m,k)
- Bob receives the ciphertext, and applies key and D(c,k) to *decrypt*, resulting in the plaintext m=D(c,k)

System and attacker model, requirements

A system can only be secure wrt well-defined assumptions/models

- A *system model* that describes the involved legitimate parties, their actions and behaviour
- An *attacker model* that provides an exhaustive description of the attacker
- A list of requirements for the operation of the system, and the security requirements

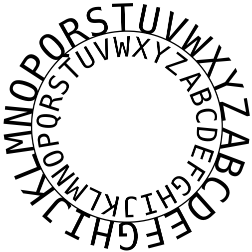
Substitution and Transposition ciphers

Basics of substitution ciphers

- Historical ciphers, used until middle of last century
- Mono-alphabetic: plaintext and ciphertext based on alphabet (A-Z)
- Bijection (complete mapping) between both alphabets

Example (Caesar's cipher)

Shift all characters by k in alphabet
For $k = 3$:
'SECURITY'$\Rightarrow$'VHFXULWB'
Shift back to decrypt. Try out here:
web.forret.com/tools/rot13.asp



Security Assessment of Caesar's cipher

- System: Alice and Bob share key, no secure channel
- Attacker: Has ciphertext, does not have key, wants plain text
- Requirements: Confidentiality of plaintext, need key to decrypt
- How to attack? Effort?

Security Assessment of Caesar's cipher

- System: Alice and Bob share key, no secure channel
- Attacker: Has ciphertext, does not have key, wants plain text
- Requirements: Confidentiality of plaintext, need key to decrypt
- How to attack? Effort?

Brute force attack

- Try all possible values for keys (only 26)
- Derive which of the plaintexts is the correct one
- How can we make attacks harder?

Improving substitution ciphers

- The keyspace of Caesar's cipher is extremely small.
- Improve by a random mapping between the 26 in/output characters
- E.g., A$\rightarrow$ X, B$\rightarrow$ D, C$\rightarrow$ M,...
- How many different mappings exist?

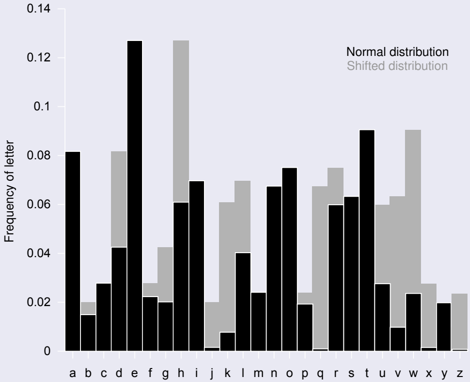
Improving substitution ciphers

- The keyspace of Caesar's cipher is extremely small.
- Improve by a random mapping between the 26 in/output characters
- E.g., A$\rightarrow$ X, B$\rightarrow$ D, C$\rightarrow$ M,. . .
- How many different mappings exist?

- 26! $\approx 4 \cdot 10^{26} \approx 2^{88}$
- But there are better ways to attack than brute force

Frequency analysis of ciphertext

Frequency of letters in English



Language-specific distribution can be used to identify substitutions

Advanced Substitution schemes

Examples to break up know frequency distribution:

- Have several alternative replacements for 'e', choose randomly
- Intentionally misspell or use dialect
- Insert 'red herring' characters to mislead analysis
- Treat 'et' as a new character, map it to a new symbol α
- Substitutions are still part of modern ciphers, but must operate on alphabets with uniform likelihood

Vigenère cipher

- Published in 1553 by Giovan Battista Bellaso
- Changes the substitution mapping in period pattern
- Key is a word that defines that pattern

	a	b	c	d	e	f	.
A	a	b	c	d	e	f	.
B	b	c	d	e	f	g	.
C	c	d	e	f	g	h	.
.	.	.	.	.	.	.	.

Plaintext:	`dead beef`
Key "cab":	`CABC ABCA`
Ciphertext:	`febf bfgf`

Breaking the Vigenère cipher

- Direct frequency analysis will not be successful any more
- Frequent character "peaks" are distributed
- Solution: as key has fixed length and is repeated often:
  - Guess a key length n
  - Compute distribution for each n'th character
  - For the right key length, you will see characteristic distributions again
    - For incorrect length, distributions should be uniform
  - For correct n, derive each character of the key individually
  - Similar to n Caesar's ciphers used to encrypt the plaintext

Transposition ciphers basics

- Letters do not get replaced, but their sequence is changed
- Shared key determines new sequence
- With message "This is secret" and "bar" as password:

key	B	A	R
order	2	1	18
text	T	H	I
	S	I	S
	S	E	C
	R	E	T

The ciphertext is "HIEETSSRISCT"

- How to attack this?

Character encodings (ASCII)

In practise, data is not represented by the Latin alphabet

- Kerkhoffs already mentioned telegraphs (Morse code)
- Computing systems use binary representations, e.g. ASCII
- ASCII represents 128 Latin & control characters in 7 bits
- Example: 0x61=a, 0x41=A, "Hello"=0x48656C6C6F
- From now on, we will operate on binary data (=integers)

Substitutions on binary data

How can the substitution principle be applied to binary data?

- inversion, 2 different keys possible (one encrypts as plaintext!)
- every two bits are replaced, 4! possible keys
- $2^n!$ possible keys
- Depending on the character coding and n, some blocks might still be more frequent
- This would enable attacks again

Modern ciphers

Overview Modern Ciphers

- Ciphers operate on streams or blocks
- Stream ciphers operate on single characters at a time
- Blocks have fixed length, are processed in one go

- Mostly XOR, shifts (performance reasons)
- Some ciphers use algebraic operations such as (+*^), x mod n
- All operations are operating on finite sets of numbers

- Symmetric: same key for enc,dec
- Asymmetric: different keys for enc,dec (aka public-key crypto)

Stream ciphers vs. block ciphers

- operate on single elements of the input (single characters, bits)
- Well suited for (audio) signal transmission
- Pro: low processing delay for low data rate input
- Con: Not as efficient (throughput) for high data rates in terms of computational effort

- operate on fixed length blocks of input (e.g., 256 bit)
- Well suited for packet-based communication
- Pro: Parallelization possible, higher throughput
- Con: Data has to fit blocks, padding required, lower efficiency

One-Time Pad

- Stream ciphers are very secure if long *random* key is available
  - It is impossible to recover the plaintext from ciphertext (even with infinite resources for attacker)
  - Key can only be used once
- This ideal cipher is called One-Time Pad
  - Has been used in practise, e.g. to encrypt "red" telephone line between Russia and US
- Problem: key as long as message, must be exchanged securely
  - Assumes secure channel to exchange key
  - Why not exchange message over that channel?

Why can't we brute-force OTP ciphertext?

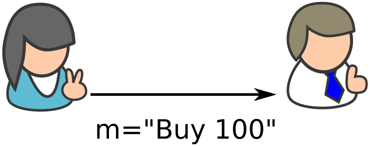
- OTP is one of the few ciphers where brute force attacks are impossible
- Brute force search through 2^n keyspace will create 2^n potential plaintexts (all possible values)
- It is impossible to determine which one was the original plaintext

Why not re-use the key?

- We could be more efficient by encrypting twice with same key?
- Example: m_1 and m_2, key stream s. $c_1 = E(m_1, s)$ and $c_2 = E(m_2, s)$
- Problem?
- As $E(m, s) = m \oplus s$,
  - $c_1 \oplus c_2 = (m_1 \oplus s) \oplus (m_2 \oplus s) = m_1 \oplus m_2$
- Bad if alphabet of m has some frequency distribution.
- Really bad if either m_1 or m_2 are known to attacker!

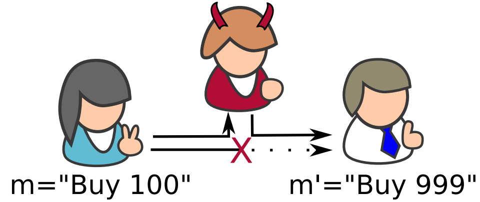
Data Integrity

Data Manipulation attacks



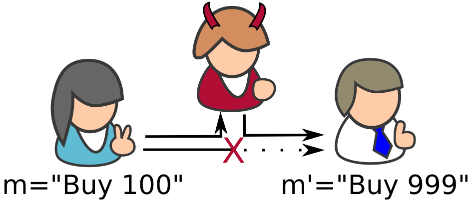
- Alice sends Bob a message:
  - "Hi Bob, I'm Alice, please buy 100 stocks of Company A"
- Alice sends the message in plaintext
- Attacker Eve wants to manipulate Alice's stock trade.
  - Eve can jam, eavesdrop and insert
- What kind of attacks are possible here?

Data Manipulation attacks



- Attack example: Attacker eavesdrops, jams, spoofs similar message:
  - "Hi Bob, I'm Alice, please buy 999 stocks of Company B"
- Bob assumes the message is from Alice, buys stocks for her
- What is the problem here?

Data Manipulation attacks



- Attack example: Attacker eavesdrops, jams, spoofs similar message:
  - "Hi Bob, I'm Alice, please buy 999 stocks of Company B"
- Bob assumes the message is from Alice, buys stocks for her
- What is the problem here?

- Secure authentication and integrity of the message

How to protect the message?

- Obvious idea: encrypt the message (e.g., using OTP)

Example (Using OTP to encrypt "buy100")

```
- "buy100"= 0x6275793130300a
- Key     = 0xA29C7B1E0E3AEE
- Result  = 0xC0E9022F3E0AE4
```

How to protect the message?

- Obvious idea: encrypt the message (e.g., using OTP)

Example (Using OTP to encrypt "buy100")

```
- "buy100"= 0x62757931303000a
- Key      = 0xA29C7B1E0E3AEE
- Result   = 0xC0E9022F3E0AE4
```

- Can an eavesdropper break the confidentiality of the message?
- Can an eavesdropping and injecting attacker change the content?

Does symmetric encryption protect data integrity?

No! Confidentiality does not imply integrity

Example (OTP and "buy100")

```
- "buy100"= 0x6275793130300a
- Key      = 0xA29C7B1E0E3AEE
- Result   = 0xC0E9022F3E0AE4
- mask     = 0x00000008090900 <- "buy100" ^ "buy999"
- Result   = 0xC0E90227370 3E4
- Plaintex= 0x6275793939390a = "buy999"
```

- As integrity is not protected, authenticity is also not protected

How does this attack work?

- We assume the attacker knows the message m="buy100"
- Lets assume the attacker wants to change to m'="buy999"
- mask on the last slide is the binary XOR of both strings=$m \oplus m'$
- With $m \oplus k = c$, the attacker creates $c \oplus mask = c'$,
- Decrypting c' with k yields:

$$c' \oplus k = ((m \oplus k) \oplus (m \oplus m')) \oplus k = m'$$

Other measures to protect integrity

- Block ciphers are not always enough
- We need a dedicated tool to validate message integrity

Cryptographic Hash Functions

Cryptographic properties for functions

- In cryptography, *preimage* resistance means that given $y = f(x)$
  - it is *hard* to find the input x for f to produce y
- *Second pre-image* resistance means that given x and f
  - it is *hard* to find an input x' for f such that $f(x) = f(x')$
- *Collision* resistance means that given f
  - it is *hard* to find any two inputs x, x' for f such that $f(x) = f(x')$
- *Random oracle* property: A random oracle maps each unique input to random output with uniform distribution
  - Informally: for two correlated inputs m_1 and m_2, the output of f is completely uncorrelated
- CRCs have only preimage resistance

Design goals for hash functions

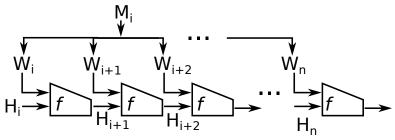
- Cryptographic hash functions are designed to have all four properties
  - Preimage resistance
  - Second preimage resistance
  - Collision resistance
  - Random oracle property
- Using cryptographic hash functions, *message authentication codes* can be constructed
- We now discuss special algorithms, similar goals can be achieved with block ciphers
- Standard hash functions are not designed to have all of these properties

SHA-1

We will explain hash functions based on SHA-1. It has the following characteristics:

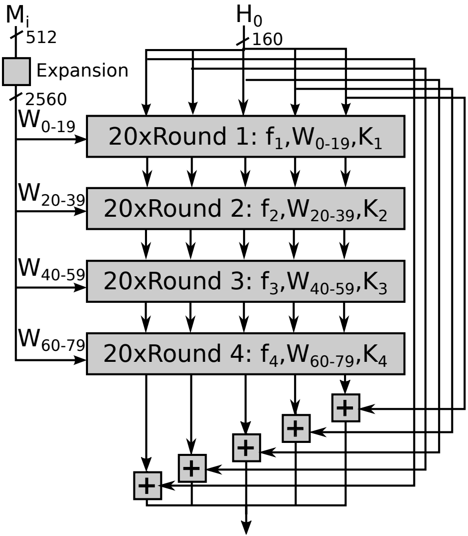
- Processes input blocks of 512 bit
- Pre-defined initial state of 160 bit
- Hash output is a 160 bit block
- Uses Merkle-Dåmgard construction
- 80 internal rounds in total

Merkle-Dåmgard construction

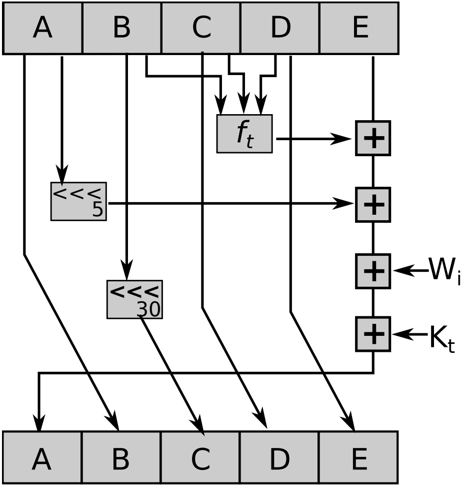


- Merkle-Dåmgard is a construction for cryptographic hashes:
    - Repeated application of a collision resistant compressing function
    - Each stage uses previous output and new chunk of input
- In SHA-1
    - SHA-1 has a constant (public) initial values in the MD chain
    - 512 bit input blocks are expanded into 2560 bit=80$\cdot$ 32 bit words
    - 4 stages, each stage has 20 rounds of compression
    - Each stage has different constants K_t and a non-linear function f_t

Overall SHA-1 operation



One round in SHA-1



Why 80 rounds?

- Increasing the number of rounds has several benefits:
  - It makes brute force attacks more expensive (each hashing takes longer)
  - It makes attacks relying on *differential cryptanalysis* harder
- The exact value for SHA-1 was most likely chosen as compromise between effort and security
- For SHA-2, 64 rounds are default. Attacks have been found for 52 round versions

Cryptanalysis of hash functions

- Two potential goals for attacker: find preimages or collisions
- Collisions are much easier to find, but less useful
- It has been shown that for MD, if f is collision resistant, then H is collision resistant
- Attacking the collision resistance of f is a first part of attack
  - Find two plaintexts that hash to the same value
  - What is the estimated effort for an n bit hash? 2^n?
- Actually, it is only $2^{\mathrm{n}/2}$. Why?

Birthday paradox:

- What is the probability, that in a group of n people, two have the same birthday?
- Variant: for which group size, the probability approaches 0.5?

- for 23 people, the probability is 50%
- for 70 people, the probability is 99.9%

- For SHA1, a minimum hash length of 160 bits is usually suggested
- A 160 bit has relates to 2^{80} effort to find collision (considered infeasible today)

How to use birthday attack for an attack

- Collisions can be directly be used to attack
  - Commitment schemes
  - Digital signature schemes
  - TLS certificates (more on them later, breaks TLS)
- Anything where the plaintext is under direct control of attacker
- Attacks have been demonstrated for MD5 (precursor of SHA-1) and SHA-1
  - Keywords: "MD5 Collisions Inc" and SHAttered
- Birthday paradoxon does not help for second preimage finding
  - Our message authentication system can use SHA-1 safely

Cryptanalysis of SHA-1

- In Feb 2005, researchers found the following:
  - Collisions can be found with effort 2^{69} steps (instead of 2^{80}, factor 2048)
  - In 2009, that result was claimed to be improved to 2^{52} steps (but found to be incorrect)
  - If assuming 2^{60} tries required, and 2^{14} ops per SHA-1 [1]
  - Nowadays, breaking SHA-1 would probably cost
    - In 2015, \$700k
    - In 2018, \$173k
    - In 2021, \$43k...
- Google computed first collision in 2017[2], claim took 9,223,372,036,854,775,808≈2^{63} tries
  - 6,500 years of single-CPU computations and 110 years of single-GPU computations.
  - Assuming 100\$ per year per CPU, cost=650,000\$

[1] schneier.com/blog/archives/2012/10/when_will_we_se.html

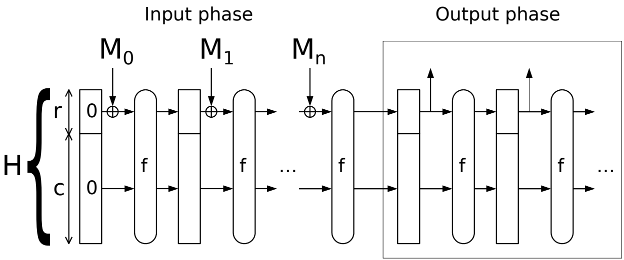
[2] https://shattered.io/

SHA-1, SHA-2, SHA-3

- SHA-2 was designed by NSA (like SHA-1), and published in 2001
- US National Institute of Standards and Technology (NIST) "promotes" security standards
- Successor of SHA-2 was chosen in a semi-public process
- In Oct 2012, Keccak was selected as SHA-3 algorithm
  - Focus on security and implementation speed
- SHA-1 appears to have weaknesses as discussed
  - SHA-2 shares a lot of the structure
- SHA-3 should be considered for high-security projects

SHA-3

- SHA-3 (Keccak) is fundamentally different to SHA-1/SHA-2
- It uses a "sponge" construction instead of MD
- r bits of message are "fed" into S per round
- r bits of output per round can be taken out afterwards



Message authentication codes

Motivation MACs

- In last lecture, we had the "buy100" example
- Alice and Bob share a key k
- Using OTP or stream ciphers, they cannot guarantee integrity
- Using SHA directly also does not help
  - Attacker can compute new hash, flip bits as well
- Message authentication codes prevent this attack

Message authentication codes (MACs)

Requirements:

- Alice and Bob share k
- Alice wants to send m to Bob, can add some x
- Using x, Bob should verify integrity of m
- Both have access to cryptographic hash function $\mathrm{H}(\cdot)$
- How to construct from k,m, and $\mathrm{H}(\cdot)$?

Message authentication codes (MACs)

Requirements:

- Alice and Bob share k
- Alice wants to send m to Bob, can add some x
- Using x, Bob should verify integrity of m
- Both have access to cryptographic hash function $\mathrm{H}(\cdot)$
- How to construct from k,m, and $\mathrm{H}(\cdot)$?

- Secret as prefix: $x = H(k||m)$
- Secret as suffix: $x = H(m||k)$
- Alice will then send (m, x) to Bob

Which is better? $x = H(k||m)$ or $x = H(m||k)$?

- One of the two allows attacker to create valid MAC for a version of m with additional blocks at the end
  - i.e. attacker can produce valid $x' = H(k, m||m')$
- One of the two allows attacker to re-use MAC if second preimage can be found
  - i.e. attacker can reuse hash: $x = H(k, m')$, iff $H(m) = H(m')$
- Can you figure out which? (Assuming MD construction)

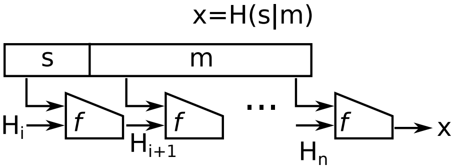
Which is better? $x = H(k||m)$ or $x = H(m||k)$?

- One of the two allows attacker to create valid MAC for a version of m with additional blocks at the end
  - i.e. attacker can produce valid $x' = H(k, m||m')$
- One of the two allows attacker to re-use MAC if second preimage can be found
  - i.e. attacker can reuse hash: $x = H(k, m')$, iff $H(m) = H(m')$
- Can you figure out which? (Assuming MD construction)

- Attack on prefix MAC: append another block to known MAC
  - $H(m||m') = H(m')$ with initial state $H_0 = H(m)$
- Attack on suffix MAC: find m' with $H(m') = H(m)$
  - Then, the original HMAC is also valid for m'

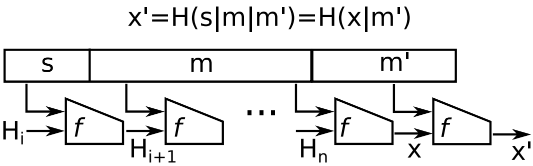
Details on attack on $x = H(s|m)$

- The attack exploits the fact that x captures the full internal state of the hash.
- Attacker can build on x to derive x'



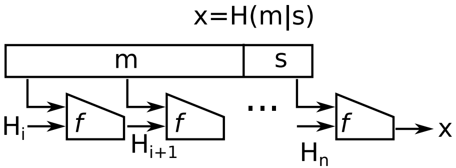
Details on attack on $x = H(s|m)$

- The attack exploits the fact that x captures the full internal state of the hash.
- Attacker can build on x to derive x'



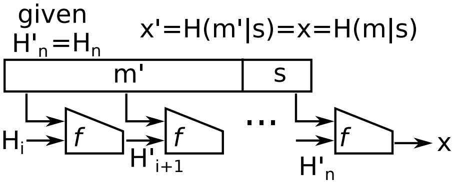
Details on attack on $x = H(m|s)$

- The attack exploits the fact x is valid for all m that hash to same value (second preimages)
- Attacker can simply re-use x



Details on attack on $x = H(m|s)$

- The attack exploits the fact x is valid for all m that hash to same value (second preimages)
- Attacker can simply re-use x



Hash-based MACs (HMACs)

- HMAC combines both prefix and suffix secrets to defeat attacks
- Construction: $\text{HMAC}(k, m) = H((k\oplus$ opad) || H((k$\oplus$ipad) || m))
  - k is a secret key padded with zeros
  - opad is outer padding (0x5c5c5c. . . 5c5c)
  - ipad is inner padding (0x363636. . . 3636)
- So, Alice sends $(m, HMAC(k, m))$ to Bob
  - Bob computes HMAC for m and k
  - Bob accepts message as authentic if HMAC is same as sent
  - Attacker cannot construct valid HMAC without k
  - Attacker cannot change m without changing HMAC

Storage of secrets

Storage of secrets

Consider the following problem

- You want to store a set of username and their passwords
  - `alice p4ssw0rd`
- Other users might be able to have read (or attackers copy data)
- How to protect the passwords of the users?

Storage of secrets

Consider the following problem

- You want to store a set of username and their passwords
  - `alice p4ssw0rd`
- Other users might be able to have read (or attackers copy data)
- How to protect the passwords of the users?

In our Linux installations, passwords are stored as SHA512 hashes in (/etc/shadow)

- When user inputs the password, it is hashed and compared with hash
  - `alice f1697e66a08b79532d5802a5cf6ffa4c`
- This is intended to keep the passwords secret
  - Can you think of ways to attack this scheme?

Finding Preimages

- Attacker has the hash values of passwords
  - Needs to find the original passwords
  - Sounds impossible?

Finding Preimages

- Attacker has the hash values of passwords
  - Needs to find the original passwords
  - Sounds impossible?

- Attacker creates a list of likely passwords
- Compares hash of each one with stolen hashes
- For short passwords, complete lists can be precomputed (Rainbow tables)
- Using rainbow tables, large sets of user/hash tuples can be processed quickly
- Example: Hashcat

Other examples for attacks on hashes

Yuval's square root attack (Collisions)

- Attacker wants Alice to sign a m text of his choice
- Attacker wants Alice to believe that she signed harmless text t
- Attacker generates n different variations m_i, t_j of m and t
- If there is one collision between m_i and t_j, attack is succesful
  - Alice checks t_j and signs the harmless text
  - The signature is also valid for m_i, as it has the same hash!

Password-based Authentication

Terminology

- Access control
  - Allow/deny users access to resources
  - Sometimes, delegation is possible
- Authentication verifies correctness of data and source
  - In this context: verifying the *identity* of login request
  - *identification* itself does not include verification

Why is guessing passwords so easy (compared to keys)?

- Passwords use string.printable
- Passwords are somewhat short
- Some passwords are used more frequently
  - Or have frequently used components
- This enables semi-intelligent brute-forcing
  - dictionaries
  - hybrid attacks

Dictionary attacks

- Users prefer simple passwords
- Dictionary attacks produce lists of popular passwords
- Ordered by popularity
- This maximises likelyhood of success with minimal tries
- Often based on sets of passwords that became public

Rank	Password
1	123456
2	password
3	12345678
4	qwerty
5	abc123
6	123456789
7	111111
8	1234567
9	iloveyou
10	adob123
...	...

Popular Passwords 2013 (according to SplashData)

Hybrid attacks

- Users have heard about dictionary attacks
- "p4sSw0Rd" might not be in dictionary
  - But it is still pretty similar
- Hybrid attacks also try combinations and popular substitutions
  - E.g. replacements such as "$a->4$" and "$o->0$", case
- Interesting estimation of effort for attacks:

`https://www.bennish.net/password-strength-checker/`

Finding Passwords in practise

- Both dictionary attacks and hybrid attacks can be used to build long lists of likely passwords
- If there is an API to submit unlimited password attempts, this could be called to break into a system
  - In practise, accounts are quickly locked to prevent this
- In most cases, dictionary and hybrid attacks are used to attempt to find preimages of hashes
  - Password hashes were stolen in some attack
  - Attacker has *unlimited attempts* to find preimage

Strengthening Passwords

Multi-factor authentication (MFA)

- Simple, most common form: Two-factor authentication (TFA)
- Combines username/password with second way, e.g. text messages
  - Example: DBS login into account
- MFA is an application of "defense in depth"

Finding Hash Preimages

Brute Forcing Hashes

- Common cryptographic hashes have length 128 (e.g., MD5), 160 (SHA1), 224-512(SHA3)
- Brute-forcing SHA1 takes about $O(2^{160})$ computations
- How could we speed this up?
  - Precompute some/all values!
- If we precompute 2^{160} hashes, we can directly look up preimage
  - Unfortunately, this takes $160 * 2^{120}$ Terabyte of storage

Improving Brute Force

- So clearly, computing hashes on demand or full precomputation is infeasible
- Can we mix both?
  - Do precomputations, but only store a subset of the found hashes
  - Just store as many hash values as you have storage space
  - Ensure that you can recover preimage of the hash values
- This is the idea behind rainbow tables, which are based on hash chains

Improving Brute Force (continue)

- In the following, we ignore *hash collisions* to simplify things
- We also assume that the *input space* is smaller than the *output space*, e.g. if only 10 character inputs are considered. . .
- Note: Rainbow tables are not computed "on the fly" to look up one hash
  - Direct brute force would be more efficient in that case
  - Rainbow tables allow you to re-use brute force effort for many hashes

Hash Chains

- Hash chains trade storage space vs. computational effort
- General operations of hash chain:
  - H() is hashing function from input domain to hash domain
  - R() is some reduction function, mapping from hash to *input domain*
    - If you only care about string.printable of length < 10, then R() should map into that
- Lets initiate a hash chain with I_1 as first input

$$I_1 \xrightarrow{H()} O_1 \xrightarrow{R()} I_2 \xrightarrow{H()} O_2 \xrightarrow{R()} \cdots \xrightarrow{H()} O_t$$

- After t operations, we get hash output O_t
- If we store I_1 and O_t, how can we find I_t?

Hash Chains

- Hash chains trade storage space vs. computational effort
- General operations of hash chain:
  - H() is hashing function from input domain to hash domain
  - R() is some reduction function, mapping from hash to *input domain*
    - If you only care about string.printable of length < 10, then R() should map into that
- Lets initiate a hash chain with I_1 as first input

$$I_1 \xrightarrow{H()} O_1 \xrightarrow{R()} I_2 \xrightarrow{H()} O_2 \xrightarrow{R()} \cdots \xrightarrow{H()} O_t$$

- After t operations, we get hash output O_t
- If we store I_1 and O_t, how can we find I_t?

- We re-compute the chain with I_1 until we hit I_t

Rainbow tables

- Rainbow tables trade time vs. space in hash reversal
- A rainbow table consists of m hash chains of length t
- For each chain, only the first input (i.e., plaintext) I_1 and the last hash O_t are stored.
- Overall space requirement: $(|I| + |O|) * m$
- Even more space-efficient:
  - Each of the m chains can use its index as starting input I
  - For each chain, only the last hash O_t is stored
  - Overall space requirement: $|O| * m$
- The product $m * t$ must be $\geq$ number of possible input values
  - E.g. if 10 characters [a-Z]: 26^{10}
- Runtime of hash lookup: $O(t/2 + t/2)$ if comparisons are free, and hashing is only expensive operation

Rainbow table operation

- How to use hash chain for lookup: We want to find $X : Y = H(X)$
- Check if Y is in list of last chain elements
  - if yes ($Y = O_z$): regenerate chain z using the input value I_z. Then find the I_z that was used to compute O_z, this is our X
  - if no: compute H(R(Y))=Y', see if this is in list of last chain elements
    - if yes ($Y' = O_z$): regenerate chain z using the input value I_z. Then find the I_z that was used to compute O_z, this is our X
    - if no: apply further iterations of reduction and hashing on Y

Rainbow table operation (continue)

- Estimated effort: (only counting hashing, as most expensive op)
  - expected $t/2$ reductions to find a matching last value + $t/2$ average effort to regenerate chain$\Rightarrow O(t/2 + t/2)$
- Brute force effort
  - Either $O(n/2 = m * t/2)$ (on-the-fly computation) or
  - $O(1)$ computation and $O(m * t)$ space

Defending against Rainbow tables

To make rainbow tables infeasible, a salt (random number) is added to each hash

- E.g.: $x = H(m||s)$ with salt s
- The salt can be stored with hashed password (x,s)
- The salt should be different for each user
- What is the benefit?
- The attacker cannot just use the same rainbow table
- If attacker would want to pre-compute rainbow tables: n bit salt increases effort for attacker by 2^n
    - Each salt requires own rainbow table of same size as original one
- Some people say rainbow tables are dead. . .

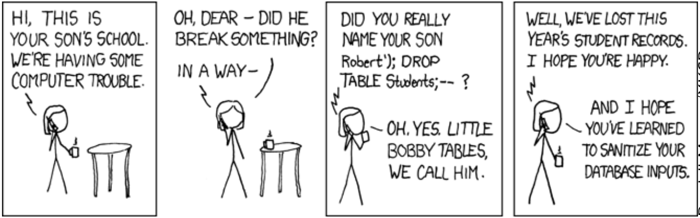
Hashcat

- If salts are used, brute force might be the only solution to find preimages
  - In particular, if non-salt part of input is from small space
  - In particular, if hashing function is computationally cheap
- Effort for the attacker directly depends on cost of hash
  - If hashing can be done in 1% of time, attack is 100 times faster
  - Bitcoin caused a lot of specialized hashing hardware to appear
  - Modern GPUs can also be used for hashing (e.g. NVIDIA CUDA)
- Hashcat is an example tool to do such online attacks
  - http://hashcat.net/oclhashcat/
  - Hashcat leverages GPUs for hashing using OpenCL

Server security: Injection (user provided input)

- Processing input from untrusted sources is dangerous
  - Buffer overflow (strings, images, ...) (upcoming lectures)
  - SQL injection
- But this is the server's main job!
- Even harder: presenting user content to users
  - Cross-site scripting (XSS)
  - Language filtering, image filtering, copyright, ...
- Example attacks using user provided input
  - Buffer overflows
  - SQL injection
  - Cross-site scripting (XSS)

SQL Injection



Randall Munroe/XKCD (CC BY-NC 2.5)

- SQL is a *query language* for databases, based on ASCII strings
- SQL injection attacks rely on incorrect validation of input data
- If user input is directly inserted in interpreted code (SQL)
  - Attackers can try to change the code
  - Could allow attacker to do anything with database

SQL injection example

- Given a SQL query with user-provided string *userName*

```
SELECT * FROM Students WHERE name = '$userName';
```

- With normal input, e.g. *userName="Robert"*, the query is

```
SELECT * FROM Students WHERE name ='Robert';
```

- With *userName = "Robert'; DROP TABLE Students;- -"*

the query is

```
SELECT * FROM Students WHERE name ='Robert';
DROP TABLE Students;--';
```

- What will be the result?

SQL injection example (Continue)

- As result of the previous attack, all student entries would be deleted
- Could also be used to read out content from other tables
  - Especially if all results of SQL query are returned to the user
  - Attacker will have to learn about table names etc first
- We will look at that in the lab.

SQL injection countermeasures

- Restriction (+validation) of character set for values
  - E.g. only a-z, A-Z, 0-9
- Proper escaping of special characters
  - E.g. turn ' into " and so forth, troublesome
- Semantic analysis of query for execution
- Restrictive configuration of database
- Disallow dropping or selection on sensitive data
- Use of prepared statements (with parameterized queries). On example:
  SELECT * FROM Students WHERE name = ?;
  - More parameterized query examples: `https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Query_Parameterization_Cheat_Sheet.md`

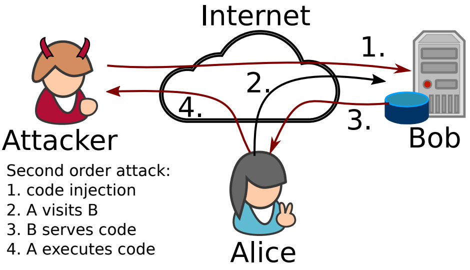
Server security: XSS

- In Cross-Site Scripting (XSS) attacks, user content enables attacks on other users

Cross-Site Scripting (XSS)

- In XSS attacks, a server sends out the script of an attacker
- Target is the user, not web server. Code executed by browser.
- Dangerous, as the user *trusts* content from known websites
  - Browsers also use *origin-policies* to restrict access to one site
  - These policies can be bypassed with XSS attacks
- Enabled by improper validation/escaping of user data
- XSS types:
  - Persistent/ second order XSS attacks
  - Reflected/ First order XSS attacks

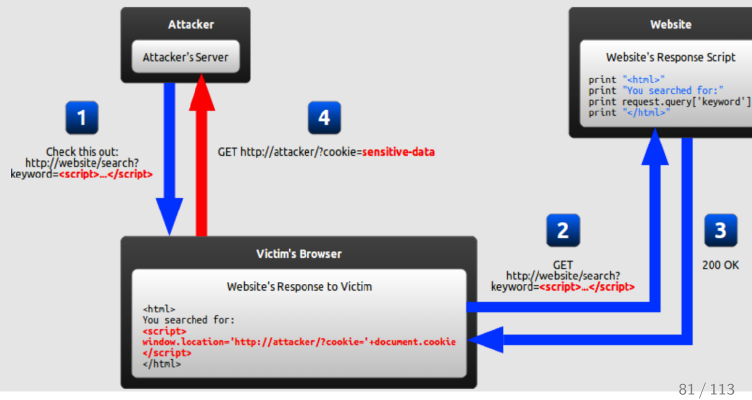
Persistent/ second order XSS attacks



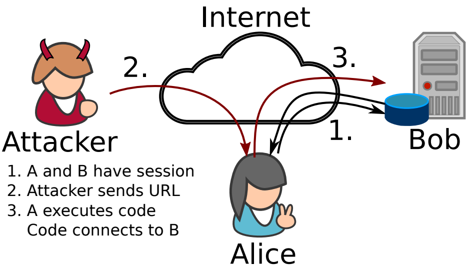
- Code will be **stored** by the server, and sent out from now on.
- Attack delivery method: Upload attack, users who view it are exploited
- Example: "Samy" Myspace worm

Reflected/ First order XSS attacks

- Non-persistent code injection. Server **reflects** back the injected code.
- Attack delivery method: Send victims a link containing XSS attack
- One example:



Cross-site request forgery (CSRF)

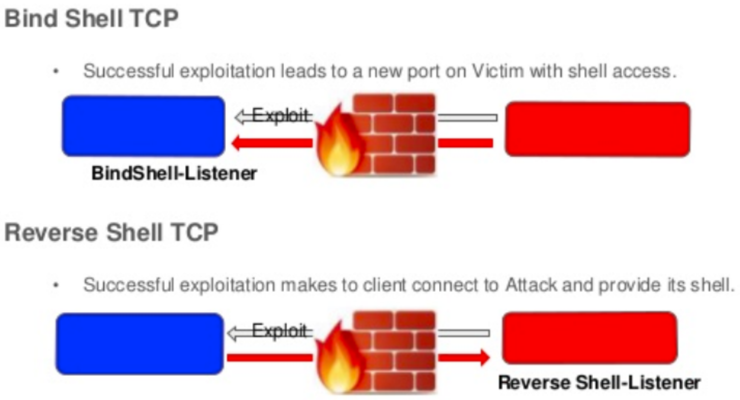


- Similar to XSS, CSRF attacks are injecting code (e.g. via URL)
- **XSS abuses users' trust in server, CSRF abuses server's trust in user**
- The code in CSRF is the executed by the victim's browser
  - Connects to third party server (e.g. facebook)
  - Uses existing autenticated session with that server

Command Injection

Reverse Shell: when needed?

A reverse shell is an attack technique used when the target machine is not directly reachable (due to firewall, NAT, etc).

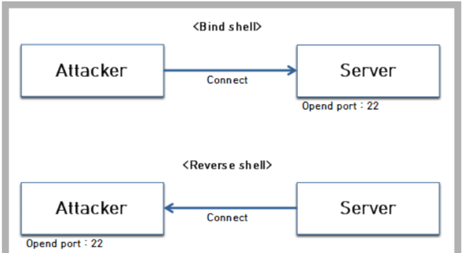


Reverse Shell: What is it?

- Bind shell : A shell that **the attacker uses after connecting to the server**. A bind shell is **setup on the target host** and binds to a specific port to listen for an incoming connection from the attacker.
- Reverse shell: A shell that **the attacker uses after the server connects to the attacker**. A reverse shell is a shell **initiated from the target host** back to the attacker who is in a listening state to pick up the shell.

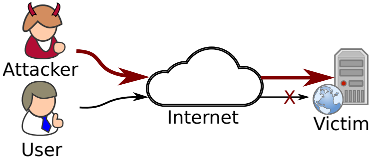
You can open a reverse shell using Netcat (exercise in the lab) or other tools.

Note: Port 22 in the picture below can be any other unfiltered port.



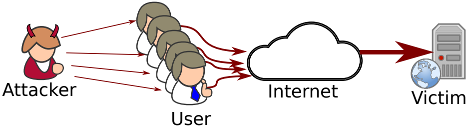
Server security: Denial of service attacks

Denial of service (DoS)



- Any server on the web has limited bandwidth (HW/link)
- *Denial of Service* (DoS) attacks exhaust this bandwidth
- Example: simple DoS use ICMP ping flooding
  - Target receives large number of pings, replies to each

DDoS attacks



- DoS attacks need high bandwidth for attacker
- *Distributed DoS* allows multiple uplinks
  - Often relies on Botnets or many users (anonymous/LOIC)
- Similar attacks are possible with many other protocols (TCP SYN, DNS, ...)
  - Ideally, high *amplification* of attacker's effort for the victim

Social Engineering

- Spearphishing is an example of a *social engineering* attack
- Any attack in which the attacker tries to trick the user in performing an action
  - Spoofed emails that tell customers to send payment to different account
  - Phone calls
  - Mail
- Usually pretending to be person of authority, or in need of help
- Inherent lack of authentication in real-world is a problem

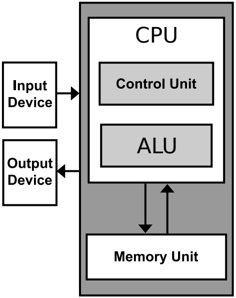
Buffer Overflow attacks

Overview Buffer Overflow

- Are a major attack vector to inject code
- Originally exploited data/code mix in von Neumann memory
  - Data provided by attacker could be executed in place
  - Control flow data (return address) next to user data
  - Some functions allow user to write more data than intended
- Buffer overflow attacks exploit well-known vulnerabilities
  - In particular, related to C langue
  - In particular, related to important insecure functions in LibC
- The overflow overwrites data on the stack, which will influence control flow
  - Most importantly, the stored *return address*
  - Other variables can also be overwritten
  - Stack frame could also be manipulated

Refresher: Architectures

- Computers commonly use *von Neumann* architecture
  - Important for us: memory holds both *data* and *instructions*
- This fundamentally enables a series of attacks
  - Attackers are able to write data over legitimate instructions
  - Attacker's data is then executed as instructions
  - Attacker could also overwrite data structures of running code
- Most prominent attack: *stack-based buffer overflow attacks*
  - But also: heap overflow, . . .

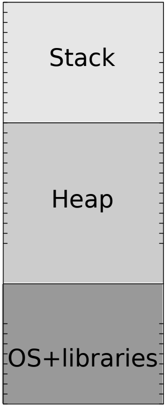


Von Neumann Architecture

Memory Layout

- Computers have different memory layers: registers, cache, RAM, discs
- Each process is presented with an abstract linear memory space to use
- OS takes care of translating memory access to the caches, RAM, disc
- Linear memory space is divided in sections
  - Stack is used for static allocation
  - Heap is used for dynamic allocation

High addr.

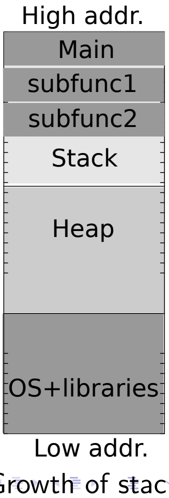


Low addr.

Linear memory space

Memory use by processes

- Processes use stack to store data structures known at *compile-time*
  - Compiler knows size, can reserve appropriate memory area in advance
- The heap stores dynamic datastructures
  - Size or number of datastructures not known in advance
  - Memory space for that data has to be dynamically allocated (malloc)
- Each subfunction call will add new memory space on stack for that subfunction
  - As result, stack use is growing with deeper function call nesting
- If a subfunction returns, its memory space is *freed* (not erased)



Calling conventions

- Stack space for each function is called *stack frame*
- Stack frame header is prepared by calling function
- *Calling convention* defines the way that each stack frame header is organized
- This way, caller and callee can be sure that stack is in expected format
- Here, we will only discuss the System V AMD64 ABI calling convention for Linux/Mac

64 bit stack layout (System V AMD64)

Registers:

- RBP: base pointer, which points to the base of the stack frame
- RSP: stack pointer, which points to the top of the stack frame
- RDI, RSI, RDX, RCX, R8, R9 used to provide arguments

To call a function:

- up to 6 arguments are passed in registers
- two more arguments/pointers can be passed on the stack
- Then, the return address is stored on stack
- Then, the calling RBP is stored on stack
- RBP is set to address of old RBP
- RSP is set to RBP-(space for variables)

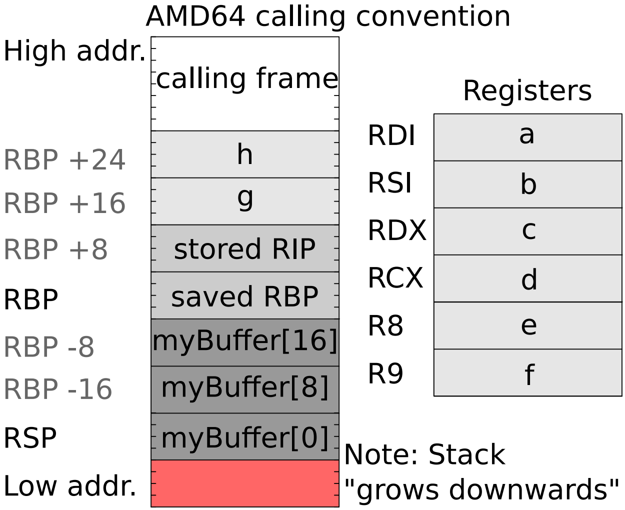
Example C code (8 arguments)

Example (Function call (C/AMD64/Linux))

```
long myFunc(long a, long b, long c, long d,
            long e, long f, long g, long h)
{
    char myBuffer[24];
    gets(myBuffer);
    return a+b+c+d+e+f+g+h;
}
```

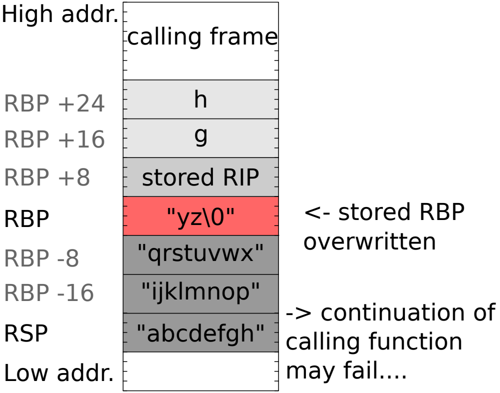
- Important points about this function
  - 24*8(=3*64) bit variables
  - 8*64 bit arguments

Example stack layout



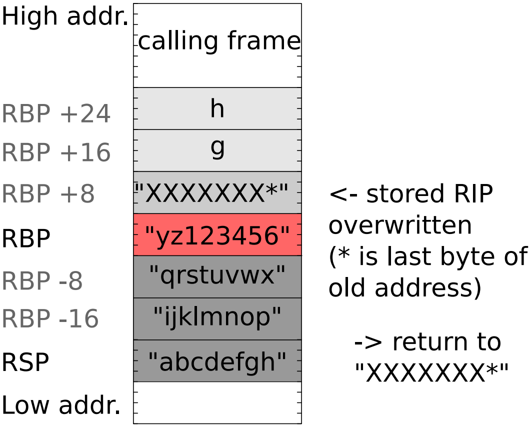
Simple Buffer Overflow

- Assume `myBuffer` contains `abcdefghijklmnopqrstuvwxyz\0`
- `gets()` continues to read in until `\0` char



Malicious Buffer Overflow

- `myBuffer=abcdefghijklmnopqrstuvwxyz123456XXXXXXX\`
- return addr. is overwritten with `XXXXXXX`

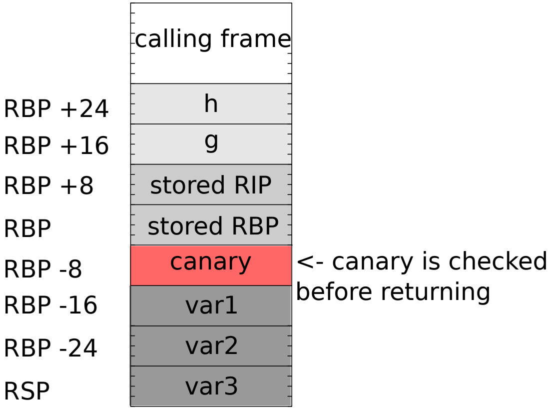


Countermeasures of buffer overflow attacks

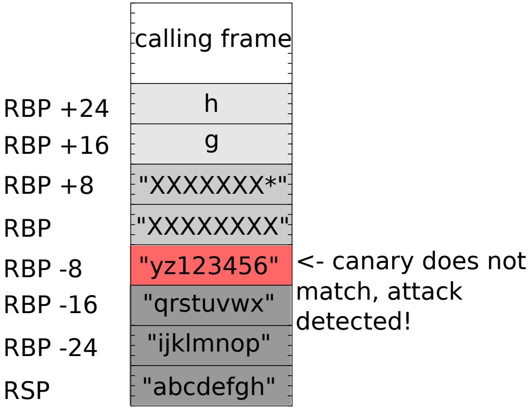
Canaries

- Canary birds were used in mines to detect gas
- Here, they are used to detect overflow attacks
- Canaries are random values saved just below RBP
- Before returning, the OS will check if the canary is "alive"
  - Canary can be random values (saved outside the frame)
  - Alternative: *Terminator* canary with \0 values, hard to overwrite
- GCC uses canaries by default! (ProPolice)
- Visual studio supports canaries as well

Canaries Figure



Canaries Figure



NX Bit

- The NX (non-executable) bit is a technology used in CPUs to **segregate** areas of memory for use by either storage of processor **instructions (code)** or for storage of **data**.
- An operating system with support for the NX bit may mark certain areas of memory as non-executable. The processor will then refuse to execute any code residing in these areas of memory.
- For example, making stack non-executable to prevent stack-based buffer overflow attacks.
- However, Return-to-LibC invariant has been proposed to defeat NX bit technology (Refer to Return-to-LibC slides later)

ASLR

- Buffer overflows require an attacker to know where each part of the program is located in memory.
- Without ASLR, libraries, stack, heap are mapped to constant addresses
- Address space layout randomization (ASLR) is an exploit mitigation technique that randomizes the location where system executables are loaded into memory (including stack address, heap address, shared library address)
- In particular when shared library address is randomized, return-to-LibC wont work since attacker needs to know LibC base address.
- Sometimes has to be enabled manually in the operating system.

Variants of buffer overflow attacks

Variants of buffer overflow attacks

- Jump into existing function (e.g. doSensitiveStuff())
- Jump into injected code by attacker
- Jump into LibC (to defeat countermeasure of NX-bit)
- Jump into PLT (Procedural Linkage Table)

We are going to focus (a bit) on the last two invariants.

Return-to-LibC attacks

- Since 2004, most OS have pages in stack either writeable OR executable...
  - NX bit, first supported by AMD64 architecture
  - So code injection does only work if NX is disabled for some reason!
- So, what can the attacker do to attack?
- NX-bit prevents jumping into injected code on stack.

Return-to-LibC attacks

- Return-to-LibC attacks return address points to LibC (standard C library) functions [3]. LibC is a library of standard functions that can be used by all C programs (and sometimes by programs in other languages)
- Addresses have to be guessed based on similar setup
- Popular functions[4] to jump into: `system()`, `unlink()`,...
- But you have to set up the stack for that function+ arguments in registers!

[3] https://linux.die.net/man/7/libc

[4] https://www.tutorialspoint.com/c_standard_library/c_function_system.htm

Return-To-PLT

- Procedural Linkage Table (PLT)
  - Used to direct executable's calls to the dynamic address of a LibC function.
- Exploiting PLT to defeat ASLR on LibC address.
- Instead of jumping into **dynamic LibC address**, we jump into **static PLT**

Malware

Types of Malware

The following are popular terms for malware

- Virus
- Worm
- Adware
- Trojans
- Rootkits / Remote Access Tools
- Ransomware

What are the differences?

Spreading classification

- Spreading by replicating code into executables
  - Viruses (uncommon nowadays)
- Spreading by automated exploit over the network
  - Worms (niche cases)
- Downloaded by the user/ browser
  - Adware (as part of *free* applications)
  - Trojans (hiding payload code as part of application)

Payloads

The payload is performing the malicious actions on victim machine

- Ad injector
- Keylogger, screengrabber, etc (Spyware)
- Rootkit
- Botclient
- Ransomware