TCP/IP (ersten 100 Slides)

Layer 1 (Physical Layer)

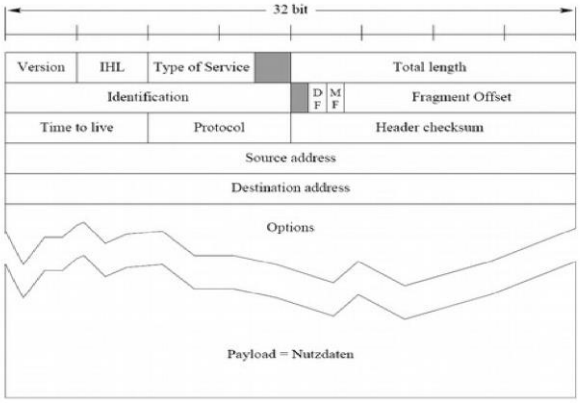
➔ Hardware (nothing you can easily control)

Layer 2 (Data link Layer)

➔ Ethernet
- Addresses: 48 bits
  - Hardwired by manufacturer
  - **MAC** address (all NICs have it)
- Type (2bytes): specifies protocol (IP,ARP,RARP)
- Data: min. 46 bytes payload (padding may be needed), max 1500 bytes
- CRC (4 bytes)

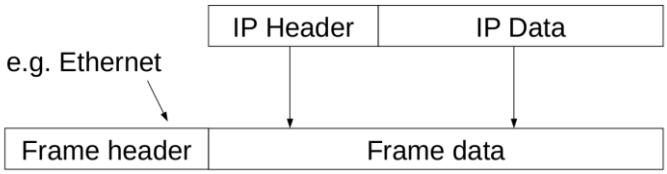
Layer 3 (Network Layer)

➔ Internet Protocol (IP)
- Is the glue between hosts of the internet
- Attributes
  - Connectionless
  - Unreliable best-effor datagram (delivery, integrity, non-duplication NOT guaranteed)
- IP Datagram:



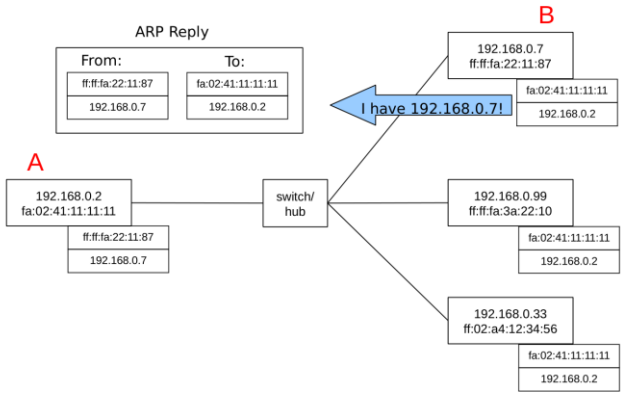
- IP Header:
  - Flags and offset (3, 13 bits)
  - Time To Live (8bits)
  - Protocol (8bits) -> UDP/TCP
  - Header checksum (16)
  - Addresses (32+32 bits) specifies source and destination

➔ Direct IP delivery
- Hosts directly connected on local network
- Problem:
  - Link layer uses 48 bit Ethernet addresses
  - Network layer uses 32 bit IP addresses
  - We want to send IP datagram but can only use Link layer (2) to do this
- Encapsulate IP datatgram in Ethernet datagram
- Encapsulation + direct delivery (to MAC):



Address Resolution Protocol (ARP)

- ➔ Host A wants to know the hardware address associated with IP of B
  - Send broadcast ARP msg on physical link layer
  - B answers A with ARP answer message



Fragmentation is used when datagram size is larger than MTU (maximum transmission unit). Each fragment will be delivered as a separate IP datagram

Layer 2/3 attacks

Fragmentation attacks:

- ➔ Ping of death (Teardrop attack) -> violates maximum IP datagram size
- ➔ IP fragment overwrite (fools the firewall) -> data is sent fragmented and the header, including the port, will be overwritten -> after packet has been reassembled, it will be delivered to the new port
  - Countermeasuers: re-assemble IP datagram on Firewall,sanity checks on IP Header

LAN attacks:

- ➔ Goals
  - Information recovery
  - Impersonate host
  - Tamper with delivery mechanisms
- ➔ Methods.
  - Sniffing
    - Eavesdrop on shared communication
    - Also possible at switched Ethernet
      - MAC flooding (switch maintains table with MAC addresses -> overflow, some switches revert to hub mode)
      - MAC duplication / cloning
  - IP spoofing
  - ARP attacks

- ARP poisoning
    - ARP does not need authentication
    - Racing against the queried host is possible
    - Fake ARP queries
    - Bot can result in redirection of traffic to the attacker
    - Used for
        - MITM attack
        - DoS
        - Target gateway (impersonate gateway to filer ALL traffic)
        - Targeted at a single host
- Countermeasures:
    - Static ARP tables on LAN
    - Drop ARP replies that have not been requested
    - Deny delivery if MAC is registered on multiple ports

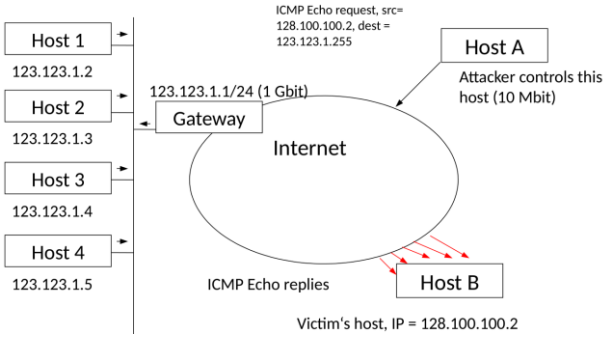
Hub is a physical layer device and forwards all packets to all ports. Switch is a link-layer device that forwards incoming **broadcast** packets to all ports but keeps track of which Ethernet addresses can be reached through which ports.

ICMP (internet control message Protocol)

- ➔ Used to exchangej control/error msg about the delivery of IP datagrams
- ➔ ICMP msg are encapsulated inside IP datagrams
- ➔ ICMP can be
    - Requests
    - Responses
    - Error messages

ICMP Echo attacks

- ➔ Information gathering: map the hosts of a network
    - ICMP echo datagrams are sent to all hosts in subnet
    - Attacker collects the replies and determines which hosts are alive
- ➔ Packet amplification (SMURF attack)
    - Send spoofed with victims IP) ICMP echo request to subnet
    - Victim will get ICMP echo replies from every machine



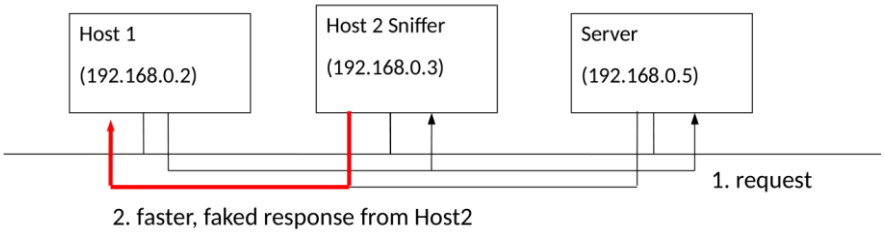
    - Countermeasuers
        - Should not work on real networks (except in the LAN)
        - Firewall configuration (allow outbound requests and inbound replies)

ICMP Destination Unreachable

➔ ICMP msg used by gateways to state that the datagram cannot be delivered (network /host / Protocol / Port unreachable etc…)

IP Spoofing

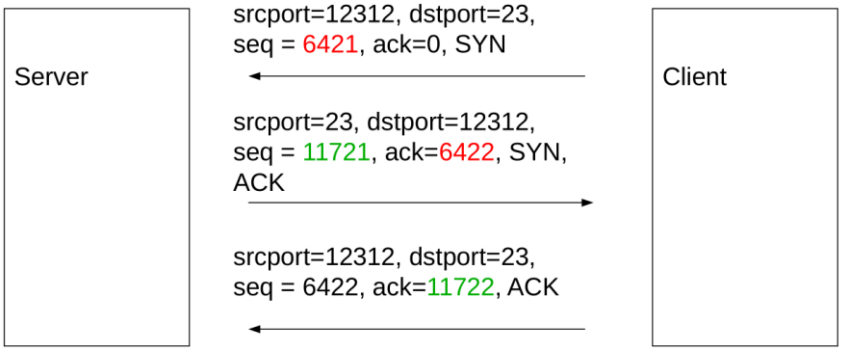
➔ Impersonating another host by sending a datagram with a faked IP-address (layer 3 attack)
- IP addresses are NOT authenticated
- Used to impersonate sources of security-critical info



Layer 4 (Transport)

➔ UDP (connectionsless, unreliable, no guaranteed delivery, implements port abstraction)
- UDP spoofing (send spoofed UDP request so that server replies to trusted client)
- UDP Hijacking (send spoofed UDP reply to client so that the client make a UDP request to the server and the server sends response to you)
- UDP storm (send spoofed UDP request to server. Server sends reply to other server which sends reply back and so forth)

➔ TCP (connection-oriented, reliable, no loss/duplication/correct ordering)
- Seq (Sequence number) specifies where payload in comm. Stream starts)
- Ack (acknowledgement number) specifies pos of next expected byte from comm partner
- TCP window (windows size specifies the amount of information that can be sent)
- TCP three way handshake to set up virtual circuit



- Shutdown virtual circuit by sending FIN flag
- Sample TCP connection

From	To	S	A	F	Seq-Nr	Ack-Nr	Payload
192.168.0.1	192.168.0.2	1	0	0	4711	0	0
192.168.0.2	192.168.0.1	1	1	0	38001	4712	0
192.168.0.1	192.168.0.2	0	1	0	4712	38002	0
192.168.0.2	192.168.0.1	0	1	0	38002	4712	‚Login:\n‘ 7
192.168.0.1	192.168.0.2	0	1	0	4712	38009	‚s‘ 1
192.168.0.1	192.168.0.2	0	1	0	4713	38009	‚e‘ 1
192.168.0.1	192.168.0.2	0	1	0	4714	38009	‚c‘ 1
192.168.0.1	192.168.0.2	0	1	0	4715	38009	‚\n‘ 1
192.168.0.2	192.168.0.1	0	1	0	38009	4716	0
192.168.0.1	192.168.0.2	0	0	1	4716	38009	0
192.168.0.2	192.168.0.1	0	1	0	38009	4717	0

- TCP Scanning
  - Portscan -> send SYN packet, if SYN/ACK received the port is opened
  - Send FIN packet (most systems return RST if closed and ignore if opened)
- OS Fingerprinting (allows to determine OS of host by examining reaction to uncommon packets)
- NMAP -> supports IP scans UDP/TCP portscans, OS fingerprinting
- TCP spoofing/hijacking is possible, very hard (bec attacker needs to DOS victim and guess correct SEQ while 3-way handshake…)
- TCP DoS attacks
  - SYN flooding (start three way handshake and stay silent, host can only keep a limited number of TCP connections in half opened state -> no more connections accepted)
  - Process table attack
  - Daemons are programs that listen on port of new connection request
    - When new connection daemon starts new process
    - Daemons often run with root
    - No new processes can be created

Internet Applications (ab Slide 100)

DNS

- ➔ Handles names to IP addresses, each domain managed by a name server, mostly UDP
- ➔ Root DNS is responsible for resolving .at
- ➔ Root name servers (13 around the world, top level hierarchy .org,.com,.at etc)
- ➔ A server that cannot answer a query, forwards query up in hierarchy
- ➔ DNS data contains questions, answers , authoritative info

Simple DNS Spoofing

- ➔ When authentication is performed an DNS names with reverse lookup
- ➔ DNS query is forwarded to DNS server for IP address that logs in (under attacker control)
- ➔ This DNS server replies with faked trusted name

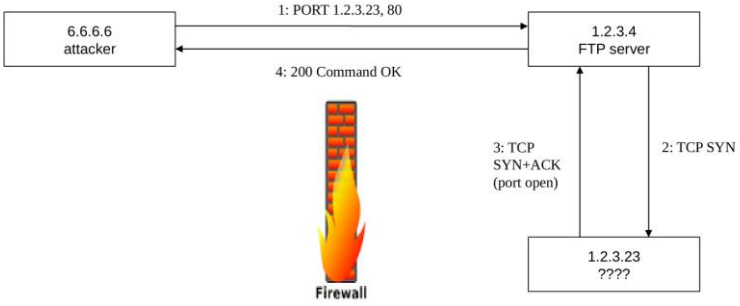
DNS Cache Poisoning

- ➔ DNS requests/replies sent over UDP
  - Reply not authenticated
  - May be spoofed

- Race against legitimate reply
- Effects:
  - Redirect traffic
  - DoS
  - MITM
  - Redirect email

FTP

- ➔ Based on TCP
- ➔ Client send connection request on server who is listening on port 21
- ➔ Client provides username and password and uses GET and PUT commands
- ➔ FTP bounce:
  - PORT command tells server address and port to be used
  - Can be used to perform TCP portscan
  - It is possible to scan behind firewall



1.2.3.23 has a server running on port 80!

  - Can also be used to send arbitrary data to ports
    - Upload data to ftp server (PUT mydata)
    - PORT destination IP, destination port
    - GET mydata

SMTP (Simple Mail Transfer Protocol)

- ➔ Push protocol
  - Used to send email
  - Used to exchange emails between servers
  - Clients have to retrieve emails via other protocols such as IMAP or POP
  - No Authentication
    - Everyone can connect to a SMTP server and transmit msg
    - Server cannot check sender identity
  - Reason: open, distributed system
    - You can receive email from anyone on the internet
    - No central authority
    - Reason for SPAM
- ➔ SMTP authentication
  - IP address (check if user is inside example.com network)
  - Extensions (SMTP-AUTH, POP-before-SMTP)
- ➔ MTA Encryption
  - POPS / IMAPS / SMTPS (SSL/TLS)

SPAM / PHISHING

- ➔ Gather destination email addresses
    - Brute force guessing
    - Harvesting
    - Malware
- ➔ Delivering spam messages
- ➔ Phishing
    - Exploits openness/weakness of SMTP protocol and Humans (social engineering aspects)

Unix Security

System call

- ➔ Performs a transition from user mode to privileged (kernel) mode

Kernel vulnerability

- ➔ Leads to complete system compromise
- ➔ Attacks performed via system calls

Kernel Integer Overflows

- ➔ FreeBSD procfs code
- ➔ Linux brk()

Linux Memory Management

- ➔ Mremap() and munmap()

Device driver code is particularly vulnerable (most drivers run in kernel mode)

Code running in user mode is always linked to a certain identity

User is identified by username (UID) and group name (GID) and authenticated by password

User root (superuser, sys admin, special privileges, cannot decrypt user passwords)

Process management

- ➔ Process
  - Implements user-activity
  - Entity that executs piece of code
  - Has its own execution stack, mem pages, file descriptor table
- ➔ Thread
  - Separate stack and program counter
  - Share mem pages and file descriptor table
- ➔ Process Attributes
  - Process ID (PID) (unique)
  - User ID (UID)
  - Effective user ID (EUID) used for permission checks
  - Saved user ID (SUID) to temporarily drop and restore privileges
  - Lots of management information

User authentication

- ➔ Login with password which are used as keys for crypt() function
- ➔ Password cracking (dictionary attacks)
- ➔ Shadow passwords
  - Pwd file is needed by many applications to map user ID to user names
  - -> /etc/passwd is readable by user (encrypted pwds not stored there)
- ➔ /etc/shadow
  - Holds encrypted passwords
  - Account information
  - Readable only by superuser and privileged programs

Group Model

- ➔ Users belong to one or more groups
  - Primary group
  - Additional groups
  - Possibility to set group pwd
- ➔ /etc/group
- ➔ Special group “wheel”
  - Protect root account by limiting user accounts that can perform su

File System

- ➔ File tree
  - Primary repo of information
- ➔ File system object
  - Files, directories, sockets, device files
  - Referenced by inode (index node)
  - “everything is a file”
- ➔ Access control
  - Permission bits
  - Chmod, chown, chgrip, umask

Type	r	w	x	s	t
File	read access	write access	execute	suid / sgid inherit id	sticky bit
Directory	list files	insert and remove files	stat / execute files, chdir	new files have dir-gid	files only delete-able by owner

SUID programs

- ➔ Each process has real and effective user / group ID
  - Usually identical
  - Real IDs determined by current user, login, su
  - Effective IDs determine right of a process, system calls
  - Attractive target for attacker

Shell

- ➔ Core unix application
- ➔ Command language and programming language
- ➔ Provides interface to Unix OS
- ➔ Rich features

Shell Attacks

- ➔ Environment variables
  - $HOME and $PATH can modify behaviour of programs
  - $IFS – internal field separator
    - Used to parse tokens
  - Preserve attack, kenn mich nicht aus
- ➔ Control and escape characters

- Can be injected into command string

➔ Applications that are invoked via shell can be targets as well

➔ Restricted shell
- More controlled environment

➔ System(char *cmd)
- Function called by programs to execute other commands
- Invokes shell
- Executes string argument
- Makes binary program vulnerable to shell attacks

➔ Popen(cahr *cmd, char *type)
- Forks a process, opens a pipe and invokes shell for cmd

File descriptor attacks

➔ SUID program opens file

➔ Forks external process (sometimes under user control)

➔ On-execute flag (if not set, new process inherits file descriptor)

Resource limits

➔ File system limits
- Quotas
- Restrict number of storage blocks and inodes
- Hard limit (can never be exceeded)
- Soft limit (can be exceeded temporarily)
- Defend against resource exhaustion (DoS)

➔ Process resource limits
- Number of child processes, open file descriptors

Signals

➔ Signal
- Simple form of interrupt
- Asynchronous notification
- Can happen anywhere for process in user space
- User to deliver segmentation faults, reload commands
- Kill command

➔ Signal handling
- Process can install signal handlers

➔ Security issues
- Code has to be re-entrant (no global data structures, atomic modifications)
- Race conditions
- Unsafe library calls, sys calls

➔ Secure signals
- Write handle as simple as possible
- Block signals in handeler
- Call only asynchronous-safe functions

Shared libraries

➔ Library

- Collection of object files
- Included into (linked) program as needed
- Code reuse

➔ Shared library
- Multiple processes share a single library copy
- Save disk space
- Save memory space
- Used by virtually all Unix applications

➔ Static shared library
- Address binding at link-time
- Not very flexible when library changes
- Code is fast

➔ Dynamic shared library
- Address binding at load-time
- Code is slower
- Loading is slow
- Uses procedure linkage table (PLT) and global offset table (GOT)
- PLT and GOT entries are very popular attack targets

➔ Management
- Stored in special directories
- Manage cache with ldconfig

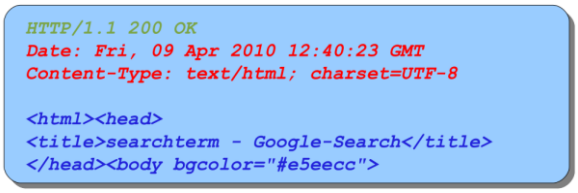
➔ Preload
- Override with other version
- Can also use environment variables for override
- Possible security hazard

Web Application Security [1]

HTTP – HyperText Transfer Protocol

- ➔ To talk to web apps
- ➔ HTTP transactions follow same general format
  - 3-part client request/server response
    - Request or response line
    - Header section
    - Entity body
  - Example:
    - Client GET request:

    - Server response:



HTTP alone usually not enough to create web apps -> scripting languages (PHP, Perl, Python …)

OWASP – Open Web Application Security Project

- ➔ Created Top Ten vulnerabilities:

A1 – Injection (SQL injection – data sent by attacker is being interpreted as commands)

A2 – Broken Auth. And Session Management (Application functions related to auth. And session management are not implemented correctly allowing attackers to comporomise pwd, session tokens or exploit implementation flaws to assume other identities)

A3 – Cross Site Scripting (XSS) (occur when application takes untrusted data and send it to a web browser without proper validation and escaping. It allowe attackers to execute script in victims browser which can hijack user sessions or redirect the user to maliciouis sites.)

A4 – Insecure Direct Object References (Dev exposes a reference to internal implementation object like file or database key. Without access control he can manipulate these references to access unauthorized data)

A5 (2013) – Security Misconfiguration (A5 (2004) – Buffer Overflows)

A6 – Sensitive Data Exposure (when web apps encrypt or hash sensitive data)

A9 (2004) - DoS

Unvalidated input (many web apps rely on client side validation which are easily bypassed)

```
system("ls $arg");
```

Protection against unvalidated input:

- ➔ Validate agoinst data type (string, integer…), allowed character set; min max length; if null is allowed, if param is required or not… regex

SQL injections

- ➔ Attacker must find a parameter that the web app uses to construct a db query.
- ➔ Attacker can obtain, corrupt, destroy db contents

```
$sql_command = "select * from users where
   username='$username' and password='$password'";
```

Obtain information using Errors

- ➔ Errors returned by app might help attacker

Blind SQL injection

- ➔ Add statements and see if the same data is returned

```
SELECT title, description
FROM   pressReleases WHERE id=5 AND 1=1
```

- ➔ If there is a vulnerability the press realease with id=5 is returned
- ➔ Now we can use trial and error approach like:

```
pressRelease.jsp?id=5 AND user_name()='h4x0r'
```

To find out information

Second order SQL injection

- ➔ SQL is injected into app but invoked at a later point
- ➔ Attacker sets username `john'--`
- ➔ At later point attacker changes pwd (and sets a new pwd for victim john)

```
UPDATE users SET password= …
WHERE database_handle("username")='john'--'
```

- ➔ Countermeasures:
  - Isolate app from SQL
  - All SQL statements have to be prepared statements
  - Use JDBC

File inclusion

- ➔ Allows attacker to include files not intended for inclusion (local LFI, remote RFI)

Web applications [2]

Simple parameter Injection Example

```
$directory_contents = `ls $directory`;
```

- ➔ Unvalidated input. If user enters "| cat /etc/passwd" he can access the file
- ➔ Countermeasures:
  - Do not use shell cmds directly in web scripts
  - Filter out chars with special meaning for shell (| * > < ; etc)

Session management

- ➔ HTTP is stateless which is bad for web apps (logged in?)
- ➔ That's why sessions
  - Web apps create and manage sessions
- ➔ Session data
  - Stored at server
  - Associated with unique session ID
- ➔ After session is created; client is informed about session ID
- ➔ If session tokens are not properly protected, attacker can hijack active or inactive sessions

Session ID Transmission

- ➔ Three possibilities:
  - Ecoding it into URL as GET param (bad because stored in logs, visible in browser)
  - Hidden form field (works for POST requests)
  - Cookies (preferable but can be rejected by client)

Cookies

- ➔ Token is set by server, stored on client machine (stored as key-value pair, "name=value"
- ➔ Uses a single domain attribute (therefore only sent back to servers with matching domain)
- ➔ Non-persistend cookies (only stored in memory during browser session)
- ➔ Secure cookies (cookies that are only sent over encrypted (TLS) connections.
- ➔ Cookies that include the IP address (makes stealing harder, breaks session when IP changes)

Session attacks

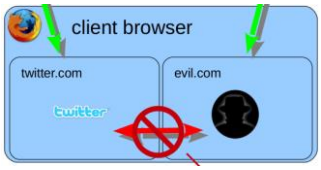
- ➔ Goal: steal session ID
- ➔ Interception: (intercept request or response and extract ID) -> use TLS for transport
- ➔ Prediction: predict or guess ID -> (possible if ID is not random e.g. username+birthday)
- ➔ Brute Force: many guesses for ID
- ➔ Fixation: make victim use certain ID

Harden Session IDs

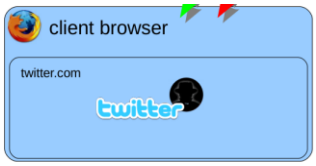
- ➔ Must be more than just unique to be secure
  - Resistant to brute force
  - Hashing session ID and encrypting has with a secret key
  - Session IDs that are truly random (for high sec apps)

JavaScript

- ➔ Embedded into web pages to support dynamic client-side behaviour
  - Client-side validation
  - Form submission
  - Etc.
- ➔ Users environment is protected from malicious JS code by sand-boxing environment
- ➔ JS programs are protected from each other by using same-origin policy to access resources



-> script will not be executed because from another site



-> script will be executed if attacker posts malicious script onto twitter and vicitim downloads content in same context as twitter

XSS

- ➔ Simple attack, difficult to prevent and can cause much damage
- ➔ Attacker can use XSS to send scripts to victim
  - Victims browser does not know if it should be trusted
  - Victim thinks it came from trusted source so it can access any cookies, session tokens or other information
- ➔ XSS categorized in two classes
  - Stored: injected code is permamently stored on the target servers (db, msg forum …)
  - Reflected: injected code is reflected off the web server (in error msg, search result or other responses that includes some or all input sent to the server as part of request)
- ➔ XSS delivery mechanisms:
  - Stored attacks require victim to browse web site (forum e.g.)
  - Reflected attacks delivered via e-mail or on other web server (user is tricked into clicking on malicious link or submitting form -> the injected code travels to vulnerable web server, which reflects the attack back to users browsers
- ➔ Likelihood of site containing XSS vulnerability is extremely high
- ➔ Countermeasures:
  - App should validate all headers, cookies, query strings, form fiels, hidden fields
- ➔ Example:

```
$query = new CGI;
$directory = $query->param("msg");
print "
<html><body>
<form action="displaytext.pl" method="get">
$msg <br>
<input type="text" name="txt">
<input type="submit" value="OK">
</form></body></html>";
```

Unvalidated input!

If script text.pl is invoked as

- *text.pl?msg=HelloWorld*

You can do whatever you want by replacing into URL like:

- *text.pl?msg=<script>alert("I 0wn you")</script>*

➔ Attacker sending information to himself by e.g. changing source of an image to www.attacker.com/+document.cookie
➔ Form redirecting to redirect the target of a form to steal form values (pwd)
➔ Length of injected script not limited
➔ XSS Mitigation (Milderung)
  - CSP (Content Sec Policy)
  - App-level firewalls
  - Static code analysis
  - httpOnly (cookie option to not allow scripting languages)
➔ Improper error handling can inform user with clues
➔ Common problem is improper handling of fail-open security check
➔ Insecure Configuration Management
  - Different server configuration problems can impact sec of site
    - Directory listing
    - Default, backup files including scripts
    - Improper file and directory permissions
    - Unnecessary services enabled like remote administration
    - Default accounts with default pwds
➔ Insecure Storage
  - Most app store sensitive information somewhere
  - Frequently encryption techniques used to protect this information
    - Devs make mistakes while integrating it into web app

DoS (Denial of Service)

➔ Attack that consumes resources to stop functionality of service
➔ DDoS
➔ Very common
➔ Zombies are poorly secured machines that are exploited
➔ Machines that control zombies are masters
➔ Web apps may be victims of flooding or vulnerability attacks
➔ Web apps cannot easily tell the difference between attack or ordinary traffic; it is difficult to filter out malicious traffic
➔ Defense is difficult and only small number of "limited" solutions exist

Testing

➔ Classification of testing techniques
- White-box testing
  - Testing implementation
  - Path coverage
  - Faults of commission (Berechtigungen?)
  - Cannot guarantee that specifications are fulfilled
- Black-box testing
  - Testing against specification
  - Only concerned with input and output
  - Faults of omissions (Unterlassung)
  - Specification flaws detected
  - Cannot guarantee that implementation is correct
- Static testing
  - Check requirements and design doc
  - Source code auditing
  - Cover a possible infinite amount of input (e.g. use ranges)
  - No actual code is executed
- Dynamic testing
  - Feed program with input and observe behaviour
  - Check certain number of input and ouput values
  - Code is executed
- Automatic testing
  - Continuously testing
  - Lot of input, output comparisons, and test runs
- Regression tests
  - Check if program has not “regressed” -> previous capabilities have not been compromised
- Software fault injection
  - Go after effects of bugs instead of bugs because bugs cannot be completely removed
  - -> therefore make program fault-tolerant
  - Failures deliberately injected into code -> effects are observed and program is made more robust

➔ Security Testing phases
- Testing must be done at all development cycle phases
  - Requirements analysis phase
  - Design phase
  - Implementation phase
  - (pre-) rollout phase

- Requirements Analysis Phase
  - Sec requirements are often not stated -> they will not be included in design and devs will not implement them

- Design phase
  - Design level (not much tool support available; formal methods; attack graphs; manual design reviews)
  - Formal methods
    - State and state transitions must be formalized and unsafe states must be described
    - Model checker ensures that no unsafe state is reached
  - Attack graphs
    - State model M, security property P
    - Attack is an execution of M that violates P
    - Attack graph is set of attacks of M
    - Attack graph generation
      - Done by hand
      - Automatic generation
- Implementation
  - Detect known set of problems and sec bugs
  - Automatic tools available
  - Target particular flaws

➔ Static Security Testing
- Manual auditing
  - Code has to support auditing with comments, architectural overview, method summaries…)
  - Microsoft -> every piece of code has to be reviewed by another dev
  - Tedious and difficult…
- Syntax checker
  - Parse source code and check for vulnerable functions (strcpy(), strcat()…)
  - Check for limited support for arguments
  - Cannot understand more complex relationships
- Annotation-based systems
  - Dev uses annotations to specify properties in source code (this val must not be NULL)
  - Analysis tool checks source code for possible violations
  - SPlint is a annotation-based systems that checks C programs for mistakes (static char bar1(/*@null@*/ char *s { return *s; })
- Model checking
  - Dev specifies sec properties that have to hold
  - Models realized as state machines
  - Statements in program result in state transition
  - Certain states considered insecure
- Meta-compilation
  - Dev adds simple system-specific compile extensions
  - They check the code
  - Can detect many bugs
- Where model checking is superior
  - Subtle errors
    - Static analysis better at checking properties in code
    - Model checking better at checking properties implied by code
  - Difference

- Static analysis detects ways to cause error
- Model checking checks for the error itself

➔ Dynamic Security Testing
- Run-time checking between OS and program
  - Intercept and check sys calls
- Run-time checking between libs and program
  - Intercept and check lib funcs
  - Often used to detect mem problems
  - Also support for buffer overflow detection
- Profiling
  - Record dynamic behaviour of apps with respect to interesting properties
- Fuzz testing (fuzzing)
  - Brute-force vulnerability detection
  - Penetrate program with lots of semi random input
  - Monitor program for crashes, dead-locks etc

➔ Security testing – (Pre-) Rollout Phase
- Prepare code for release
  - Remove debug code, debugging infos, symbols etc
  - Remove sensitive information, disable debug output (no printf())
  - Reset all security settings, remove test accounts
- Penetration testing
  - ->process of actively evaluating your information sec measures
  - Like our InetSec challenges
  - Common procedure
  - Done to assure customers security-awareness
  - Sink costs (bugs cost more)
  - Done with tools that can test particular protocol or special penetration testing suites
  - Different types of services
    - External penetration testing (traditional; testing focuses on services and servers available from outside)
    - Internals security assessment (testing on LAN, DMZ, network points)
    - Application security assessment (apps that may reveal sensitive infos are tested)
    - Wireless/remote access assessment
  - Limitations:
    - Permission to attack (client defines scope and targets beforehand; only certain systems allowed)

Buffer Overflows

One of the most widely used attacks and if successful, allows execution of arbitrary code in exploited program. The goal is to:

- Inject instructions into memory of vulnerable program
- Exploit program vulnerability to change flow of execution
- Execute injected code

Advantages

- Very effective (runs with privileges of exploited process)
- Can be exploited locally and remotely

Disadvantages:

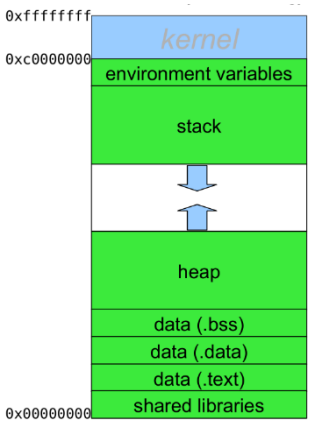
- Architecture dependent
- OS dependent
- Some guess work involved (addressing)

Memory Management

➔ Modern languages provide automatic buffer size checks when accessing mem
➔ Some languages do not provide these checks
  - Program must make sure that only allocated number of bytes are written to buffer
  - Else, adjacent memory regions will be overwritten
  - Overwritten mem regions may contain sensitive information

Memory Layout

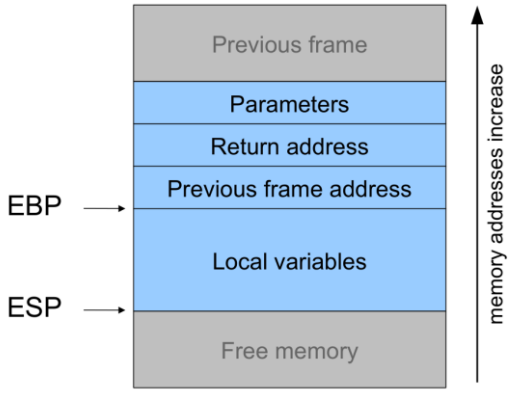
➔ Stack segment
  - Local variables
➔ Data segment
  - Global uninitialized / initialized variables (.bss, .data)
  - Dynamic variables (heap)
➔ Code (text) segment
  - Program instructions
  - Usually read-only

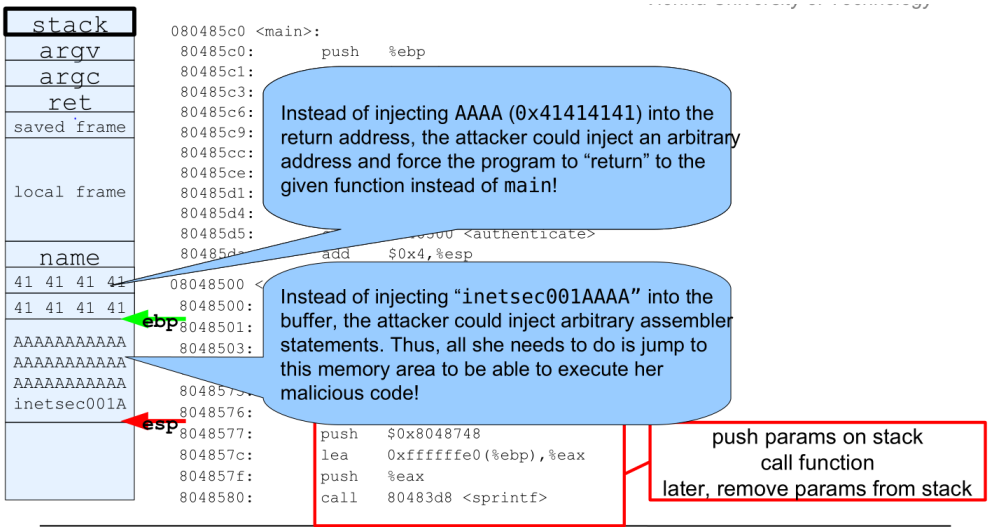


Stack

➔ Usually grows towards smaller memory addresses
➔ Special processor register points to top of stack (Stack pointer -> points to last stack element)
➔ Composed of frames
  - Upon function call, a new frame is pushed on top of stack
  - Used to conveniently reference local variables
  - When function return -> frame is discarded, last frame on stack is stored

- Address of current frame stored in processor register (frame/base pointer – FP)





Simple buffer overflow:

1. Inject some code into the process, and
2. Set code pointer to point to this content

Code pointer is most often on the return address to the calling function (alternative: func P)

The effect is, that it causes a jump to our malicious code and successfully modifies the execution flow. Out code is now executed with priviledges of running process.

Shell code

- ➔ Usually a shell should be started (execve)
- ➔ System calls (mechanism to ask OS for services)

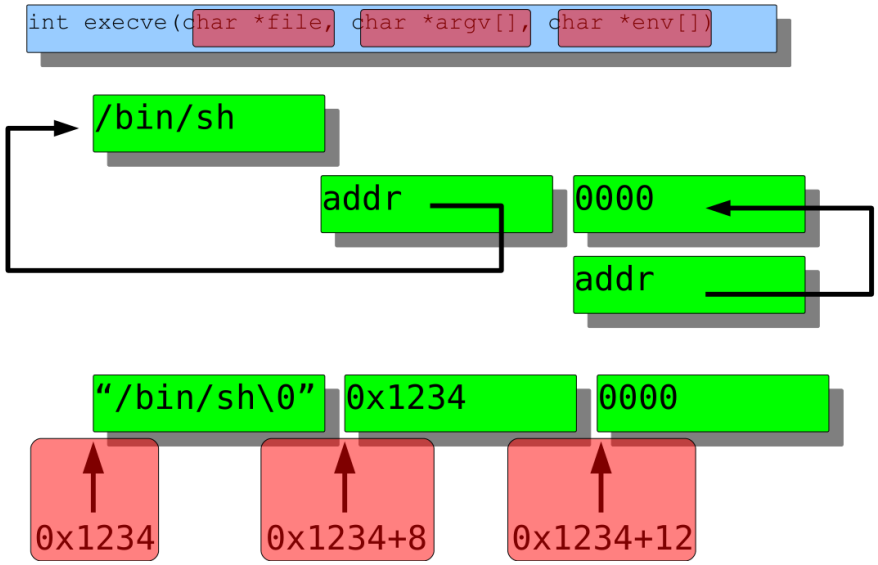
Execv

```
void main(int argc, char **argv) {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;

  execve(name[0], &name[0], &name[1]);
  exit(0);
}
```

```
int execve(char *file, char *argv[], char *env[])
```

- File is the name of program to be executed “bin/sh”
- Argv is address of null-terminated argument array -> NULL
- Env is address of null-terminated environment array -> NULL

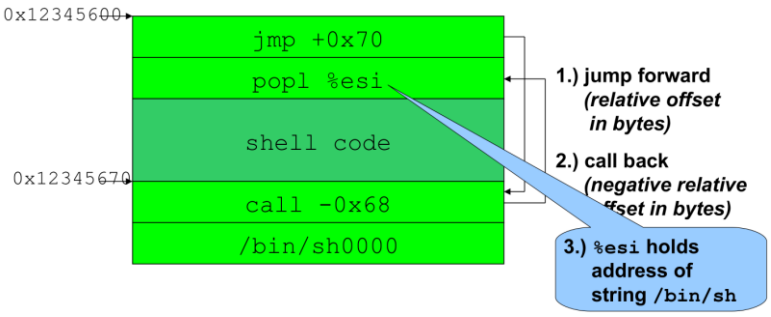


Spawning shell in assembly

1. Move system call number of execve (0x0b) into %eax
2. Move address of string /bin/sh into %ebx
3. Move address of the address of /bin/sh into %ecx
4. Move address of null word into %edx
5. Execute the interrupt 0x80

Shell code – Addressing

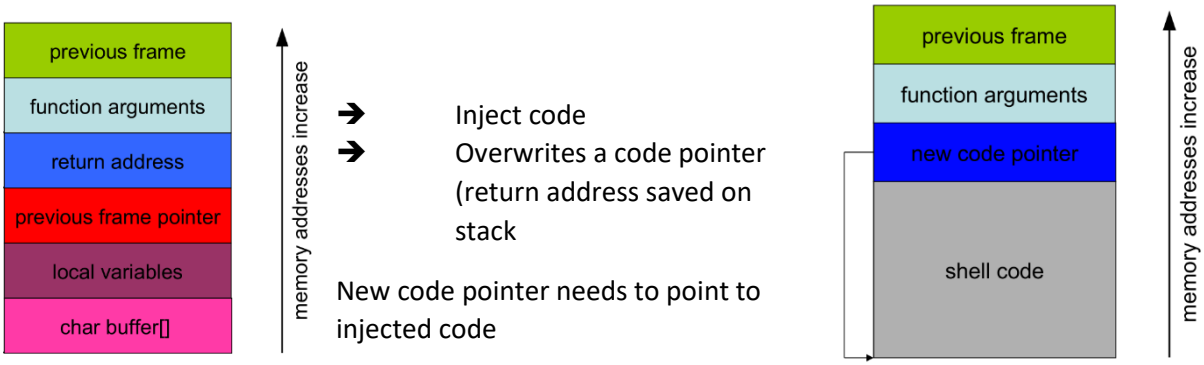
- Problem: Position of shell code in mem is unknown -> how determine add of string?
- Make use of instructions using relative addressing
  - Jmp and call (call instruction saves IP of next instruction on stack and the jumps)
- Idea:
  - Jmp instruction at beginning of shell code to call instruction
  - Call instruction right before /bin/sh string
  - Call jumps back to first instrucitokn after jump
  - Now address of /bin/sh is on the stack



Shell Code – Null Bytes

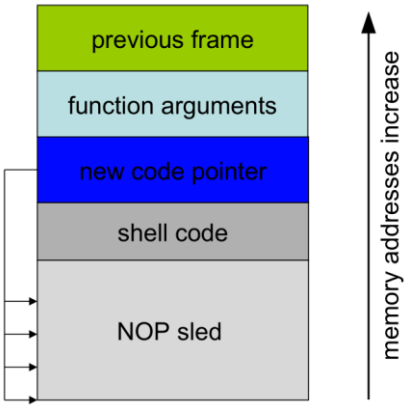
- Shell code is usually copied into a string buffer
- Problem: any null byte would stop copying -> null bytes must be eliminated

Putting it all together



Code Pointer

- ➔ E.g. return address in stack frame
- ➔ Must be overwritten with correct value
- ➔ Start of exploit code
- ➔ Has to be guessed
- ➔ NOP (no operation) sled
  - Long series of NOP (0x90) instructions at beginning of exploit code
  - Return address must not be as precise anymore



Small buffers

- ➔ Buffer can be too small to hold exploit code
- ➔ Store exploit code in environmental variable (env stored on stack)

Defenses

- ➔ Try to avoid exploitable bugs in programs
- ➔ Compiler and linker can implement some defences
    - Non-executable stack (standard BO do not work but attacker may inject in heap)
    - Address space randomization (attacker needs to know the address of that code in memory, even with NOP sled, needs to know approximately -> randomize mem layout!)
        - Different layout for each execution
        - Makes it way harder to guess addresses for exploitation
        - Defense can be broken with bruteforce on 32 bit systems
    - Stack canaries
- ➔ Use safe versions of the C standard lib functions (strcpy -> strncpy because it uses size as a parameter)
- ➔ In C++ it can be mostly avoided to use buffers

Advanced Buffer Overflows

1. Set up function parameters
2. Set code pointer to point to existing code

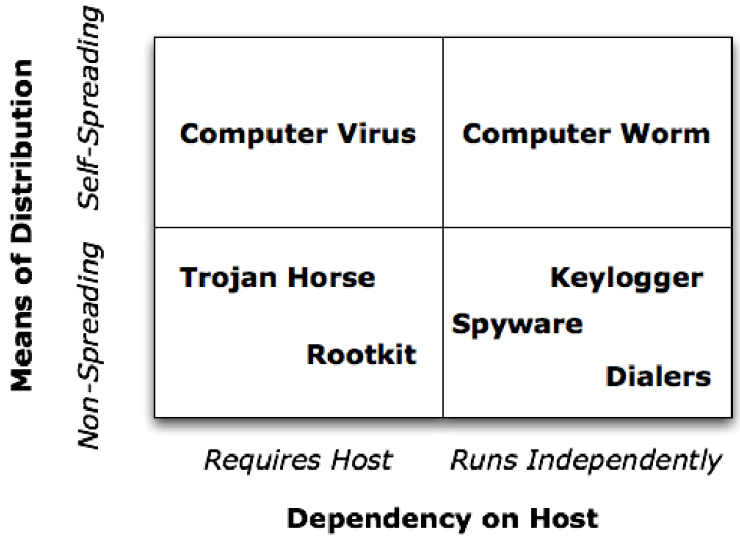
ROP – Return Oriented Programming

- ➔ Counter Measure DEP/NX in place
    - All memory regions are either executable, but read-only OR they are writable, but not executable

Malware: Offensive Programming, Botnets

Malware: software that fulfils malicious intent of author

Virus: program that reproduces its own code by attaching itself to other executable files in such a way so that its executed when the file is executed.



- Virus (self replicating, infects files)
- Worm (self-replicating, spreads over network)
- Interaction-based worms
    - Spread requires human interaction
    - Double-click on exe
    - Follow link to download exe
- Process-based worms
    - Requires no human interaction
    - Exploits vulnerability in network service, configuration, etc.

Viruses - Infection

Virus Lifecycle

- ➔ Reproduce
    - Viruses balance infection versus detection possibility
- ➔ Infection
    - Difficult to predict when infection will take place
- ➔ Attack phase
    - Deleting files, changing random data on disk…

Infection Strategies

- ➔ Boot viruses
- ➔ File infectors
    - Simple overwrite virus (damages original program)
    - Parasitic virus (append virus code and modify program entry point)
    - Cavity virus (inject code into unused regions of program code)
- ➔ Entry Point Obfuscation
    - Virus hijacks control later (after program launch)

- Overwrite Import table addresses
- Overwrite function call instructions

➔ Code integration
- Merge virus code into program
- Requires disassembly of target

Worms Propagation

Computer Worms

➔ Self-replicating
➔ Worms either
- Exploit vulnerabilities that affect large number of hosts
- Send copies of worm body via email

➔ difference to classic virusis autonomous spread over network
➔ speed of spreading is constantly increasing

Worm Components

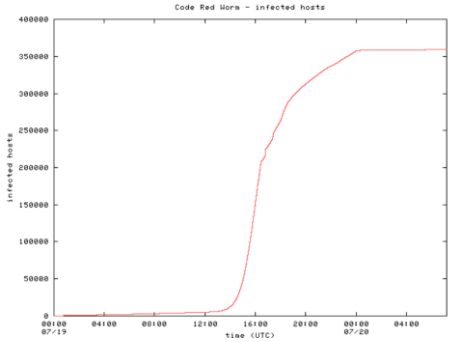
- Target locator (how to choose new victims)
- Infection propagator (how to obtain control of victim and transfer worm body to target sys)
- Life cycle manager (control different activities depending on certain circumstances, often time depending)
- Payload (today, often a botnet)

Target locator

- Email harvesting (consult address books, files might contain email addresses)
- Network share enumeration (windows discovers local computers, which can be attacked; some worms attack everything including printers -> prints random garbage)
- Scanning (randomly generate IP addresses and send probes)
- Service discovery and OS fingerprinting

Exploit-Based Worms

- Require no human interaction
  - Typically exploit network services
  - Spread faster
- Propagation speed limited by either
  - network latency (worm has to establish TCP connection – Code Red)
  - bandwidth (worm can send UDP packets as fast as possible (Slammer)
- Spread can be modelled using classic disease model
  - Starts slow, then enters exponential growth



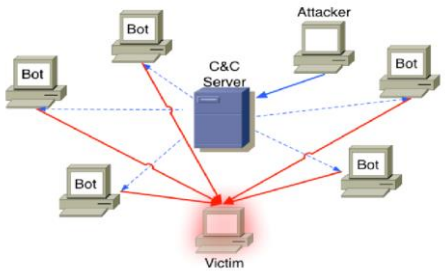
Botnets

A bot is a compromised machine that can be controlled by attacker remotely.

Features of bots:

- Remote control facility – attacker has full control
- Implementation of different commands – attack can command bots for specific purposes
- Modular design to add features dn update itself
- These features differentiate bots from worms

Botnet:



Threat posed by Bots

- For the infected host: information gathering
  - Identity data
  - Financial data
  - Private data
  - Email address books etc
- For rest of the internet
  - Email spamming
  - DoS attacks
  - Propagation
  - Support infrastructure for illegal internet activity

Bootkits

- Boot + Rootkit = Bootkit
- Infect startup code like Master Boot Record (MBR) or bootloader
- Loaded before OS Kernel (BIOS -> bootkit -> OS)

Modern bots often steal windows CD keys or keys to games, harvest email adresses, run a shell command, take screenshots, log all keypresses …

C&C Strategies (Command & Control)

C&C Arms Race

- Botnet operators try to achieve reliable C&C and protect their infrastructure
  - Centralized or decentralized C&C

Single Point of Failure

- If single C&C server it is trivial to shutdown

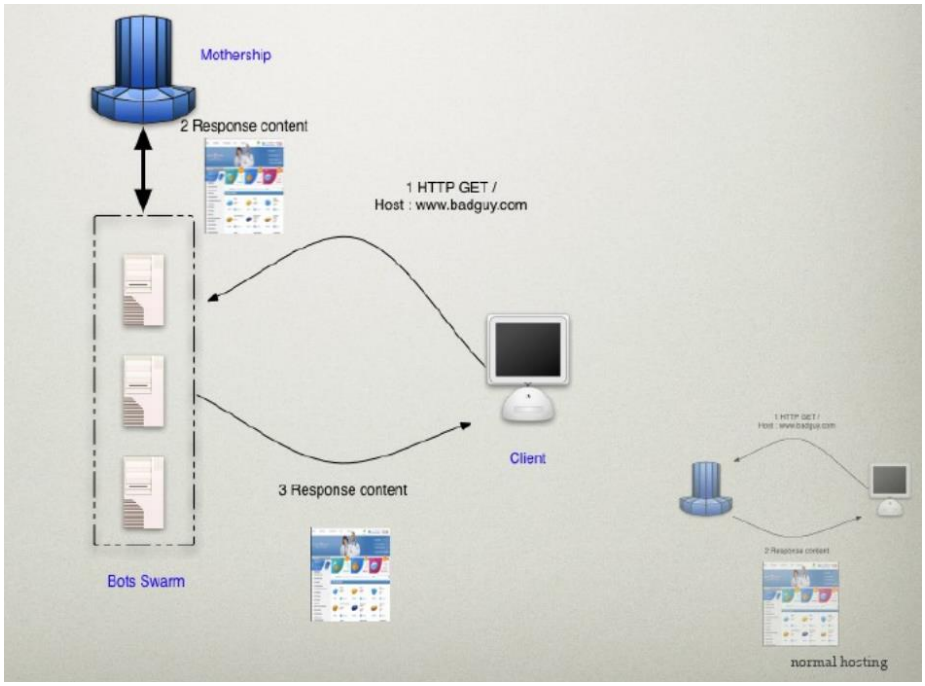
Centralized C&C Mechanisms

- Multiple C&C servers (HTTP / IRC)
- Address of C&C servers must be available to each bot
- Large botnets often partitioned int oseveral smaller ones
- IRC
    - Used because suitable to sending commands to bots
    - Default protocol in first botnets, rarely in modern ones
- HTTP
    - Outgoing HTTP allowed everywhere
    - Hard to detect unusual botnet traffic
    - State of the art C&C protocol

Bullet-Proof Hosting (hä?)

Fast-Flux hosting

- Technique used by botmasters to provide reliable hosting of services on unreliable bots
    - Network has high churn rate (bots come and go)
- Turn infected computers into providers for malicious content
- Can be used to host master servers for botnets
- Round Robin DNS
    - A DNS answer consist of several DNS A records
    - Each time the order of the answer list id different
    - Idea is to balance workload on different servers
- TTL value
    - Normally DNS records include a TTL value between 1 and 5 days for taking benefit form DNS chaching system
    - Recently between 0 and 900 seconds
- Fast-Flux
    - Several IP addresses in the DNS record
    - Low TTL value
    - Usage of round robin DNS
- DNS entries need to be frequently updated
- Single fast-flux:
    - DNSA entries frequently changed



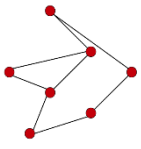
Domain Generation Algorithms (DGA)

- Bot master registers new domains for C&C on regular basis
- Bot generates list of domains and checks them for the first valid C&C server to receive new command

Decentralized C&C

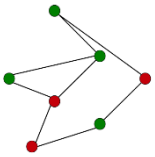
- Alternative to client-server architecture (bot commands propagate in a p2p network)
- More robust
    - Difficult to catch the botmaster
    - Harder to take down because no single point of failure
    - If some nodes shut down -> gaps are closed and network continues
- Each bot has a list of peers (neighbours) to send / receive commands

Custom P2P C&C



➔ Bots can use a custom p2p protocol
    - Designed for botnet
    - Can be very noisy (a lot of incoming/outgoing connections)

Standard p2p protocols



➔ Use standard p2p protocol
    - Allow C&C traffic to blend in with legitimate p2p traffic
    - Legitimate p2p is still blocked in many locations
➔ Build C&C on top of legitimate protocol

Takedown Strategies

- ➔ Take down master
  - Ask ISP to block IP
  - Ask .com admins to change entry for evil.com => sinkholing
- ➔ Take over master
  - Take control of IP/DNS name running the master
  - Issue commands to the botnet
    - Uninstall bot
    - Warn user he is infected
  - Legally a little risky
  - Only works if C&C is not well-protected using cryptography

Take-Downs

- Not so easy because C&C servers can be hosted by tens of different providers all over the world (hard to shut them all down; some ISPs will respond to complaints faster than others)
- For take-down to "stick", all C&C servers need to be taken down at the same time
  - If a single server is still up, it can update the bots to use new C&C servers

Sinkholing

- ➔ Is a DNS server shipping wrong information, to prevent using domain names it represents.
- ➔ Used by white hats to bock C&C domains
  - Redirect to non-existing IP or white hat C&C servers (e.g. warn infected users)

P2P Take-Downs

- ➔ Each bot has a list of peers to send/receive commands
- ➔ Poison all list entries with sinkholing IP to prevent receiving commands from botmaster
- ➔ Defense:
  - Most botnets use authenticated commands, so unsigned / wrongly signed commands are ignored
  - Prevent two IPs within the same network -> sinholder needs global distributed IPs

Examples:

- Skynet
  - Based on Zeus and uses build-in list of Tor hidden services to hide IRC C&C servers
  - Activities:
    - Information stealing
    - Bitcoin mining
  - Size: between 10.000 and 15.000 btos
  - Authors arrested 3rd December, 2013
- Carberp
  - Russia / Ukraine
- Mirai

Language Security

Postel's Law

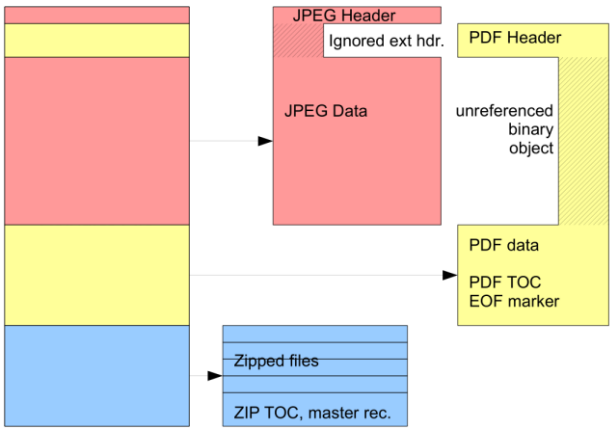
- ➔ "Be conservative in what you send, be liberal in what you accept"
- ➔ Made the internet work in early days but brings massive security implications today

Polyglot

- ➔ Source code that is valid in multiple programming languages

Binary Polyglots

- ➔ One file that is valid as simultaneously (somefile.pdf/zip/jpg)
- ➔ Example: PDF+JPEG+ZIP



Protocols are Languages

- ➔ Both have/are
  - Structure input
  - Grammar
  - Fed to a machine that parses it and reacts to it
- ➔ Protocols are often more powerful than most writers think

If your protocol is too powerful, validity is: **UNDECIDABLE**

Regular Languages

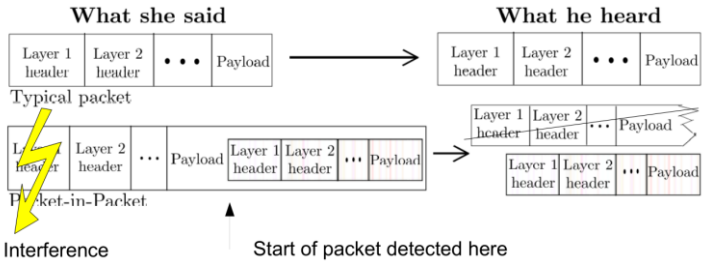
- "regular expressions"
- Finite state automata
- Simple nesting, delimiters

Turing complete = recursively enumerable

- Telling if input is a program that produces a given result: UNDECIDABLE
- Ex: telling if any given code is 'good' or 'malicious' without running it

Layer Model

- ➔ Layers
  - Encapsulate / Protection / black box (not worrying about details)
- ➔ Packets
  - Senders and receivers are certified devices
  - Receiver reads what sender transmitted
  - Noise is handled by lower abstraction layers
  - Packet in Packet: (example for modern day radios?)



Limitations

- ➔ Probabilistic attack
  - Only a fraction of packets get destroyed by interference at the right place
  - Typically, failed packets-in-packets will be ignored by victims
- ➔ Pro:
  - Attacker does not have to sit on the radio network

Works also on WiFi !!

- ➔ More complex; WiFi supports different speeds and symbols, but still possible
- ➔ Example: “Beacon in packet”

Do you trust your compiler?

- Functional equivalence != security equivalence
- Some compiler optimizations can be problematic
  - Dead storage elimination (write once, read never operations are removed)
  - Inline functions (stack frames merge, exposing private variables)
  - Alignment on stack (reduces stack entropy)

JAVA SLIDES ARE NOT PART OF THE EXAM !!!!!!!!!!!!!!!!!! :DDDD

Cryptography

Shamir's law

- Crypto is bypassed, not penetrated

Cryptographic algorithms usually not fail abruptly but gradually (unless they are not backdoored in the first place). Usually the implementation or usage is the problem. -> we need better cryptographic/security engineering

Cryptographic goals and primitives

Goals:

- Confidentiality (protect content from unauthorized access)
- Integrity (protect content from unauthorized manipulation)
- Authentication (identification of data or communicating entities)
- Non-repudiation (prevent entity from denying previous commitments or actions)

➔ Often more than one goal desired

➔ To design a secure system, you might need more than these goals

Cryptographic Primitives

- Unkeyed primitives (hash functions, real random sequences)
- Symmetric-key primitives (symmetric key ciphers, msg auth. Codes, signatures)
- Public-key primitives (public-key ciphers, signatures)

➔ Kerckhoffs principle: "A cryptosystem should be secure even if everything about the system, except the key, is public knowledge"

➔ Schneiers's law: "Any person can invent a security system so clever that she or he can't think of how to break it"

Cryptanalysis

➔ Is the study of techniques to defeat cryptographic primitives

1. Frequency analysis
2. Linear analysis
3. Related-key analysis

➔ For most primitives, there is no mathematical proof that a cryptographic primitive cannot be broken.

Attack/Threat Model

- Ciphertext only (COA) – attacker only know c
- Known plaintext (KPA) – attacker knows m and c
- Chosen plaintext (CPA) – attacker can choose m and obtain c
- Chosen ciphertext (CCA) – attacker can choose some c and obtain the corresponding m

A – Alphabet of definition A={0,1} (not further discussed in this course)

M – Message space, elements of M are called plaintext messages or just messages (m element of M)

C – Ciphertext space, set that contains strings from symbols of an alphabet

K – Key space, elements of K are called keys

Symmetric-Key Cryptography

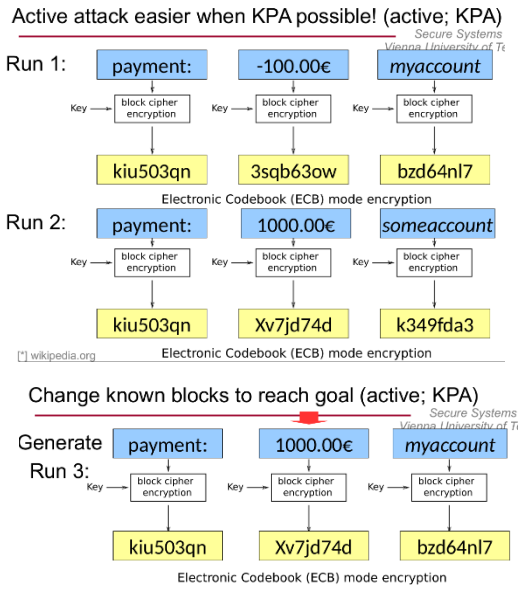
- Block ciphers
  - Break up plaintext into strings of fixed length t
  - Encrypt/decrypt one block at a time
- Stream ciphers
  - Special case of block cipher with t=1
  - However the substitution technique can change for every block
  - Key stream {k[0], k[1], k[2], …}

Block ciphers

- Product cipher
- Confusion
- Diffusion
- Encrypts blocks of fixed size
  - DES: 56 bit key, 64 bit block
  - AES: 128, 192, or 256 key ,128 bit block
- Modes of operation for block ciphers
  - Electronic Code Book (ECB)
  - Cipher Block Chaining (CBC)
  - Counter Mode(CTR)

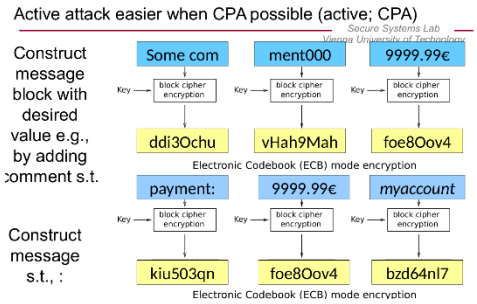
ECB

➔ Split message into blocks
➔ Pad message with random data so its length is a multiple of the block size
➔ Feed every block separately into the encryption function



-> KPA (attacker knows m and c)

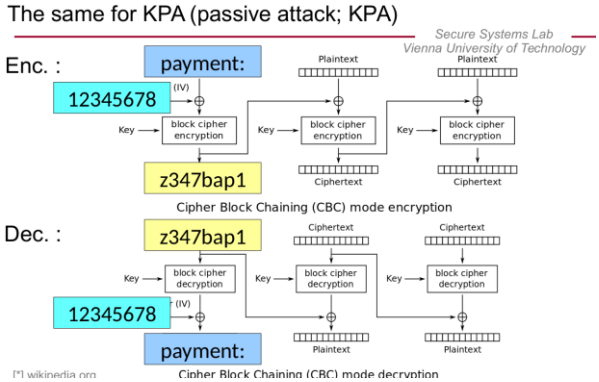
-> we want to generate cipher to get code (CPA)



CBC

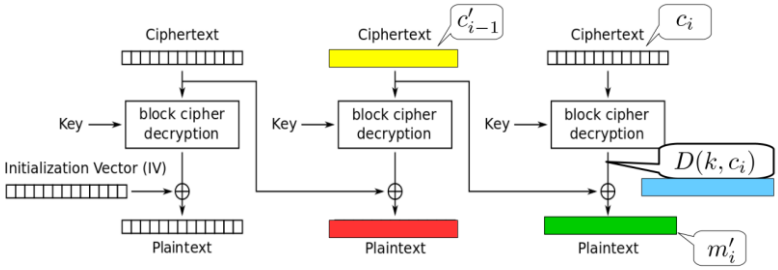
➔ Split msg into blocks
➔ Pad msg with random data so its length is a multiple of the block size
➔ Encryption of one block depends (also) on previous block
➔ Enryption of first block depends on randrom Initialization Vector (IV) per msg

➔ XOR every block with cipher text of last block before passing to encryption function (AES)



CBC Bit Flipping Attack

➔ The block containing the flipped byte will be mangled when decrypted, but corresponding byte in next decrypted block will be altered



$$m_i = D(k, c_i) \oplus c_{i-1} \qquad m'_i = c'_{i-1} \oplus D(k, c_i)$$
$$D(k, c_i) = c_{i-1} \oplus m_i \qquad c'_{i-1} = D(k, c_i) \oplus m'_i$$

➔ Intermediate state (IS) is the direct output of the encryption function D(k,ci)

CBC Padding Oracle Attack

➔ Block ciphers requires that all messages are of the same block length
➔ For the last block, padding might be required
➔ There is a known padding scheme PCKS#5 -> final block of plaintext is padded with N bytes (depending on the length of the last6 plaintext block) of value N

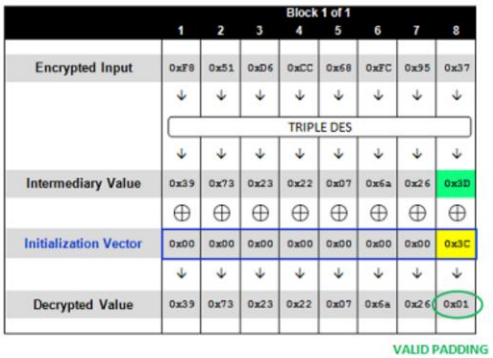
- CBC Padding example, there is always some padding

	BLOCK #1								BLOCK #2							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Ex 1	F	I	G													
Ex 1 (Padded)	F	I	G	0x05	0x05	0x05	0x05	0x05								
Ex 2	B	A	N	A	N	A										
Ex 2 (Padded)	B	A	N	A	N	A	0x02	0x02								
Ex 3	A	V	O	C	A	D	O									
Ex 3 (Padded)	A	V	O	C	A	D	O	0x01								
Ex 4	P	L	A	N	T	A	I	N								
Ex 4 (Padded)	P	L	A	N	T	A	I	N	0x08	0x08	0x08	0x08	0x08	0x08	0x08	0x08
Ex 5	P	A	S	S	I	O	N	F	R	U	I	T				
Ex 5 (Padded)	P	A	S	S	I	O	N	F	R	U	I	T	0x04	0x04	0x04	0x04

➔ Requires that 3 cases can be distinguished:
1. A valid ciphertext is received (e.g. 200 OK)
2. An invalid ciphertext is received (e.g. 500 internal Server Error)

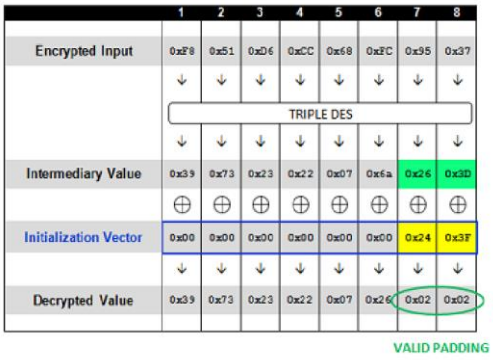
3. A valid ciphertext is received (properly padded but decrypts to invalid value) -> e.g 200 OK, &invalid Data)

➔ CBC attack reconstructs intermediate state / value (IS)



The intermediary state byte 8, is calculated as follows:

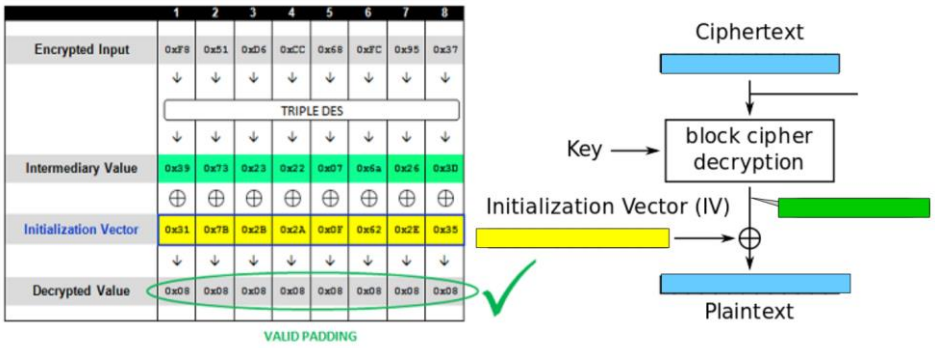
$$IS[8] \oplus IV[8] = 0x01$$
$$IS[8] \oplus 0x3C = 0x01$$
$$IS[8] = 0x3C \oplus 0x01$$
$$IS[8] = 0x3D$$



The initialization vector byte 8, is calculated as follows:

$$IV[8] = IS[8] \oplus 0x02$$
$$IV[8] = 0x3D \oplus 0x02$$
$$IV[8] = 0x3F$$

Brute force search continues at next byte IS[7]

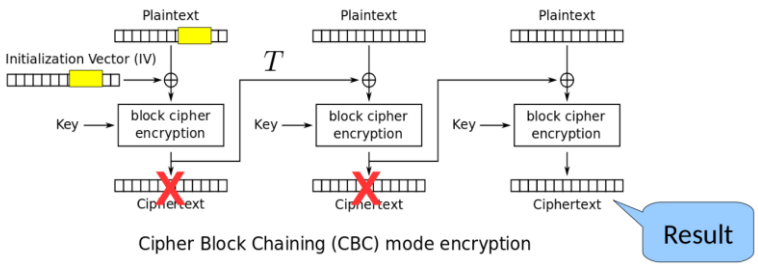


➔ Now recover plaintext of original block using original IV and brute forced IS, m = IV XOR IS

CBC-MAC

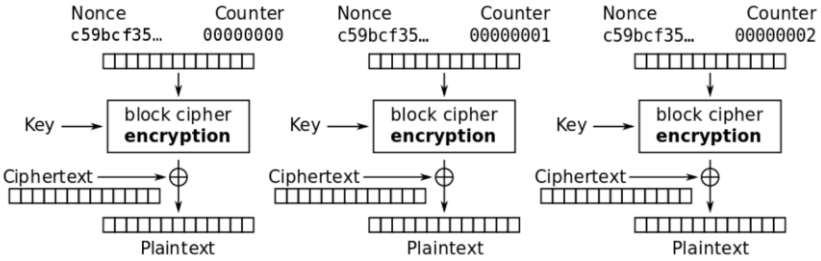
➔ Cipher block chaining Message authentication code

1. Create a msg auth code as a checksum
2. Calculation based on shared secret
3. Should ensure integrity of transmitted data



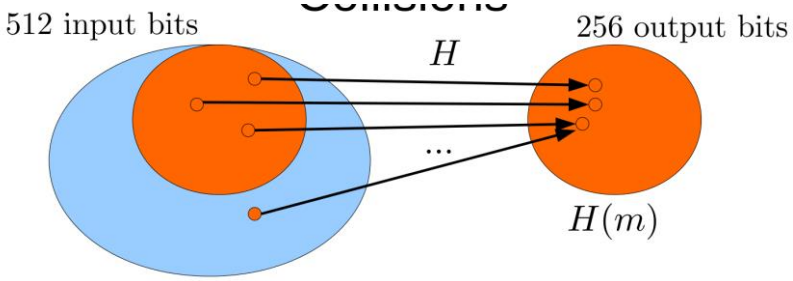
CTR

- ➔ Counter Mode
- ➔ Pad msg with random data so its length is multiple of block size
- ➔ Split msg into blocks
- ➔ Encryption of one block depends on current counter value
- ➔ XOR every block with the output of the encryption function to produce the ciphertext
- ➔ For encryption **and** decryption use:



Hash functions

- ➔ A hash function (H) takes a message (m) of any size and outputs a fixed size hash (h)
- ➔ 4 properties:
    1. Easy to compute
    2. Pre-image resistance (one-way-function: cannot go from hash to msg)
        - Hash function does not automatically hide its input if the input guessable (e.g. coint flip)
    3. Second pre-image resistance (if messages m != m' , than H(m) = H(m') should be impossible)
        - Small change in input, large change in hash
    4. Collision resistance (should not possible to find any two different messages with the same hash)



The maximum number of guesses required to find a collision is 2^256 + 1

It takes **10^27 years** to compute 2^128 hashes ….

Proof-of-work und Merkle Tree hab ich ausgelassen (wird schon nicht kommen xD)