

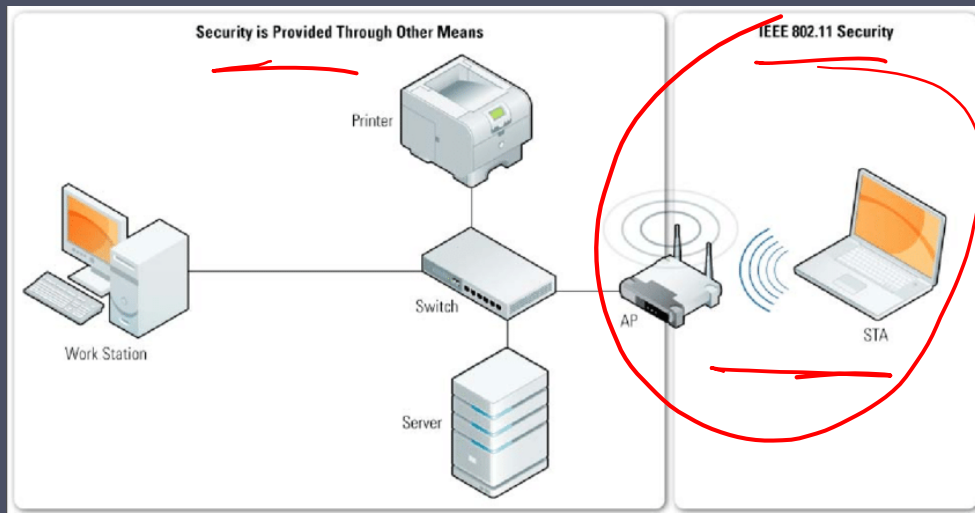
Wireless LAN Security

- The following security measures are often considered as trivial solutions for risk mitigation:
 - Disable the SSID broadcast
 - Maintain a list of authorized MAC addresses and block unauthorized ones from the network
 -  Select AP locations in the middle of the building (not close to windows), use directional antennas and line walls and windows with metal foil to minimize the signal leakage to the outside of the building
- How much security do these measures bring?
- Need something more...real security solutions

Demo AP Setting...

Wireless LAN Security

- The original 802.11 specification included a set of security features for privacy and authentication that were quite weak.
 - For privacy, 802.11 defined the Wired Equivalent Privacy (WEP) algorithm.
 - The privacy portion of the 802.11 standard contained major weaknesses.
- Subsequently, the 802.11i task group has developed a set of capabilities to address the WLAN security issues. In order to accelerate the introduction of strong security into WLANs, the Wi-Fi Alliance promulgated Wi-Fi Protected Access (WPA) as a Wi-Fi standard.
 - WPA is a set of security mechanisms that eliminates most 802.11 security issues and was based on the trimmed down version of 802.11i standard.
- The final form of the 802.11i standard is referred to as Robust Security Network (RSN).
 - The Wi-Fi Alliance certifies vendors in compliance with the full 802.11i specification under the WPA2 program.



Demo Wi-Fi Info...

Wired Equivalent Privacy (WEP) Algorithm

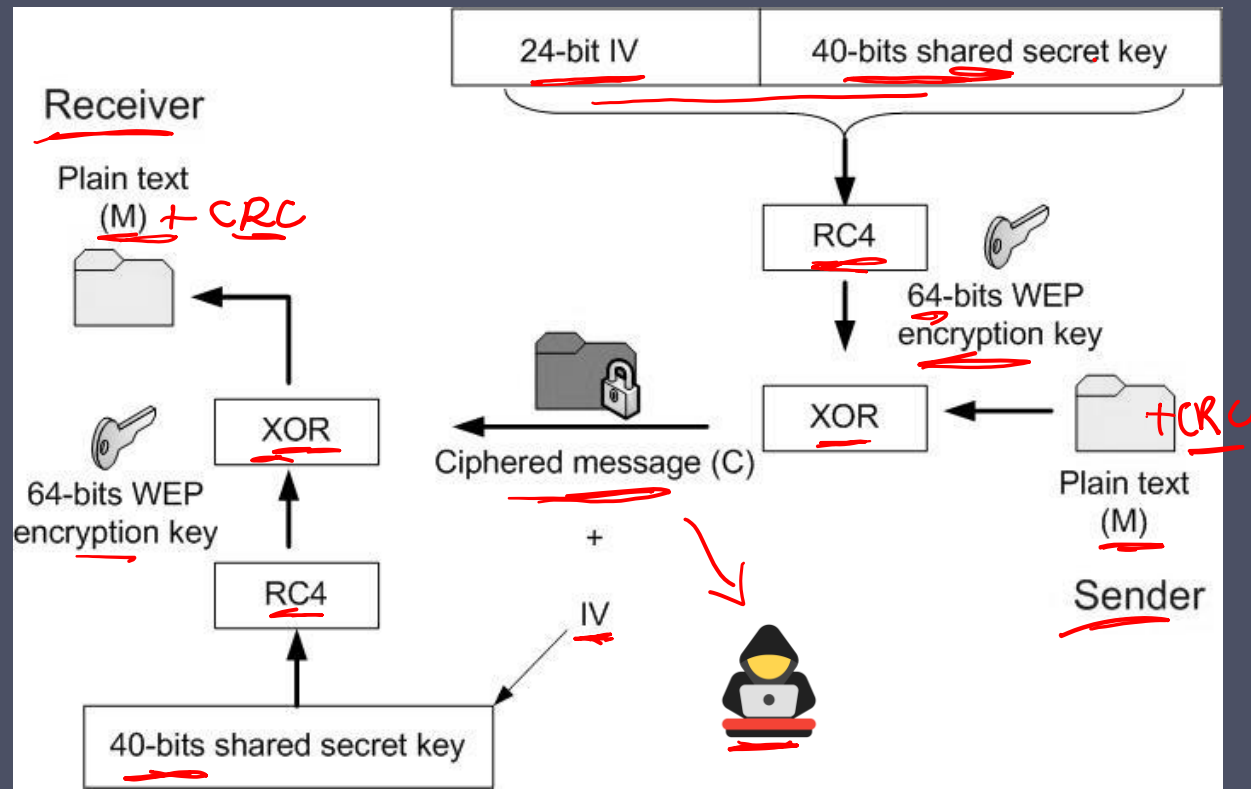
Wired Equivalent Privacy (WEP) Algorithm

WEP was part of the IEEE 802.11 standard ratified in 1997.

$2^{24} = 16M$

Provide Support for

- Authentication ✓
- Confidentiality ✓
- Integrity ✓



WEP is notoriously weak...

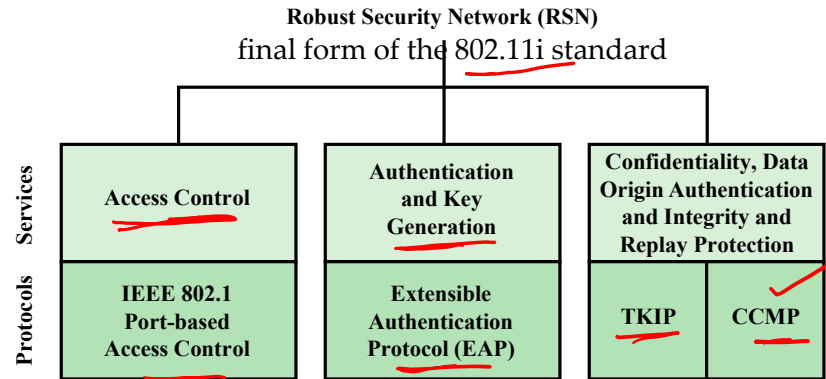
Wi-Fi Protected Access (WPA)

Brief History of Wi-Fi Protected Access (WPA)

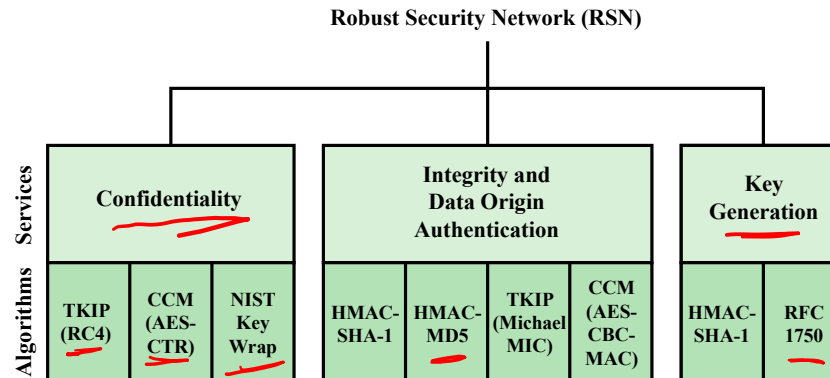
- As WEP was not secure...
- Subsequently, the 802.11i task group has developed a set of capabilities to address the WLAN security issues.
 - In order to accelerate the introduction of strong security into WLANs, the Wi-Fi Alliance promulgated Wi-Fi Protected Access (WPA) as a Wi-Fi standard.
- WPA is a set of security mechanisms that eliminates most 802.11 security issues and was based on trimmed down version of 802.11i standard...initially released in 2003.
 - No new/specialized equipment was required...just firmware update
 - Based on TKIP / CCMP for encryption
 - TKIP - Temporal Key Integrity Protocol (TKIP) ✓
 - CCMP - Counter mode CBC-MAC Protocol
- WPA2 is improved version of original WPA released in 2004.
 - Based on CCMP for encryption
- WPA3 comes as a successor to WPA2 - introduced in 2018
 - after July 2020, all Wi-Fi devices that are certified must support it.

WPA2

The Wi-Fi Alliance certifies vendors in compliance with the full 802.11i specification under the WPA2 program.



(a) Services and Protocols



(b) Cryptographic Algorithms

- CBC-MAC = Cipher Block Block Chaining Message Authentication Code (MAC)
- CCM = Counter Mode with Cipher Block Chaining Message Authentication Code
- CCMP = Counter Mode with Cipher Block Chaining MAC Protocol
- TKIP = Temporal Key Integrity Protocol

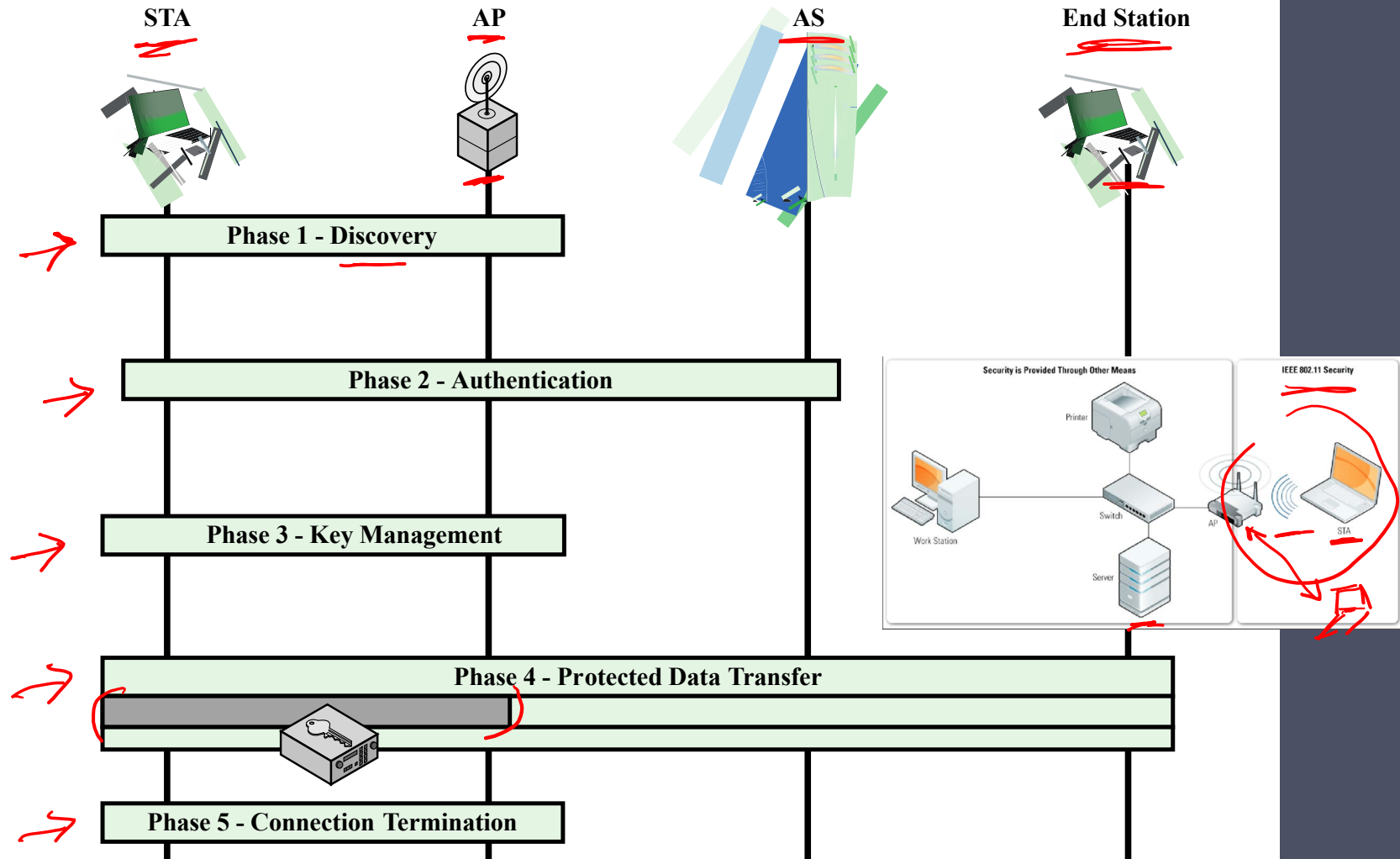
Figure 24.6 Elements of IEEE 802.11i

IEEE 802.11i Standard

Phases of Operation

IEEE 802.11i Phases of Operation

Essentially the working WPA2 protocol



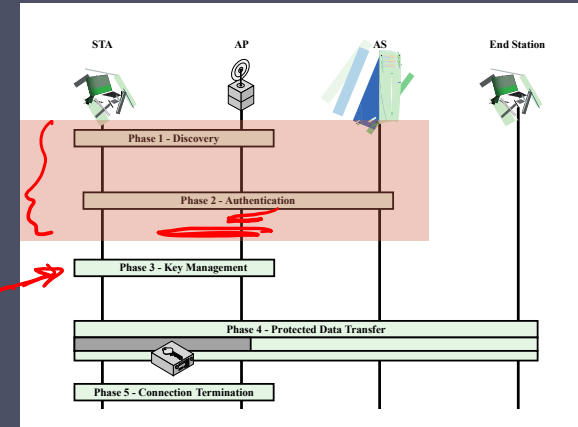
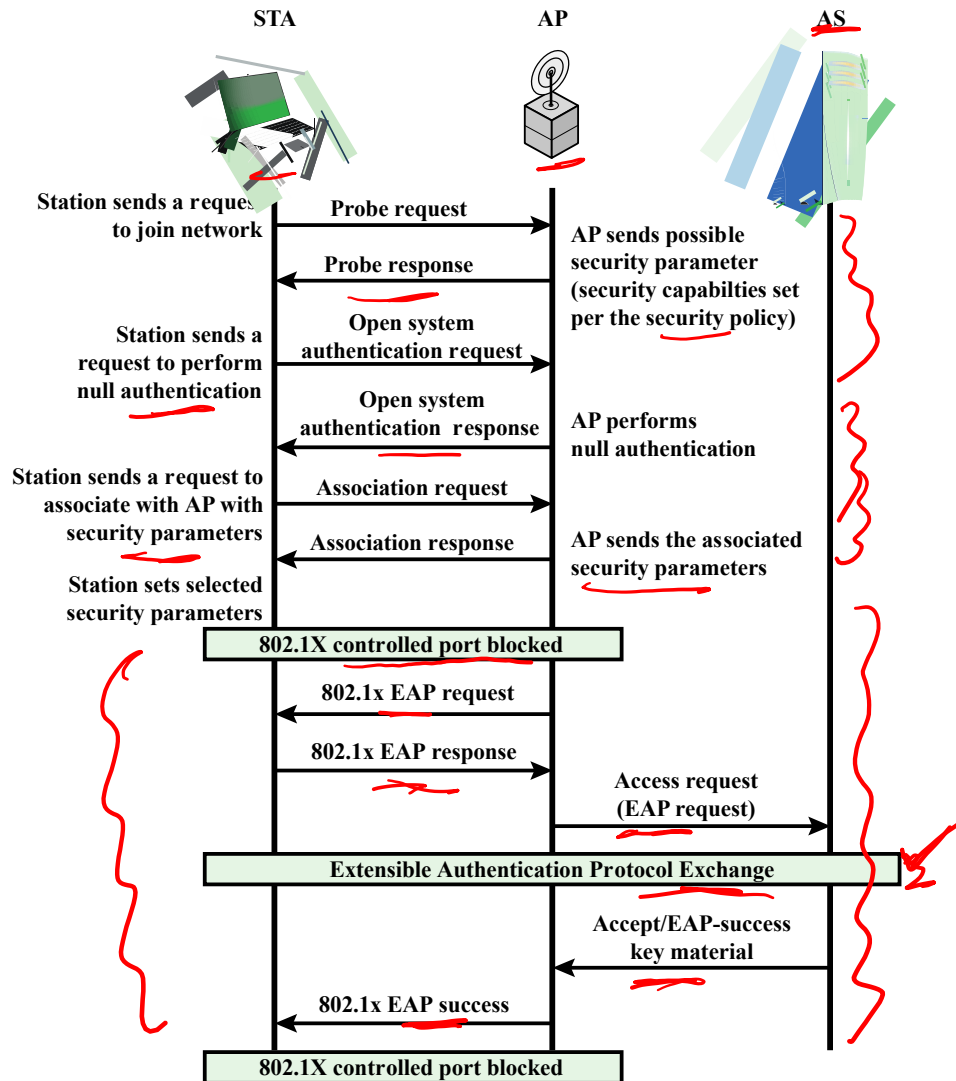
IEEE 802.11i Standard

Phases of Operation

 **Capability Discovery, Association, and Authentication**

IEEE 802.11i Phases of Operation

Essentially the working WPA2 protocol

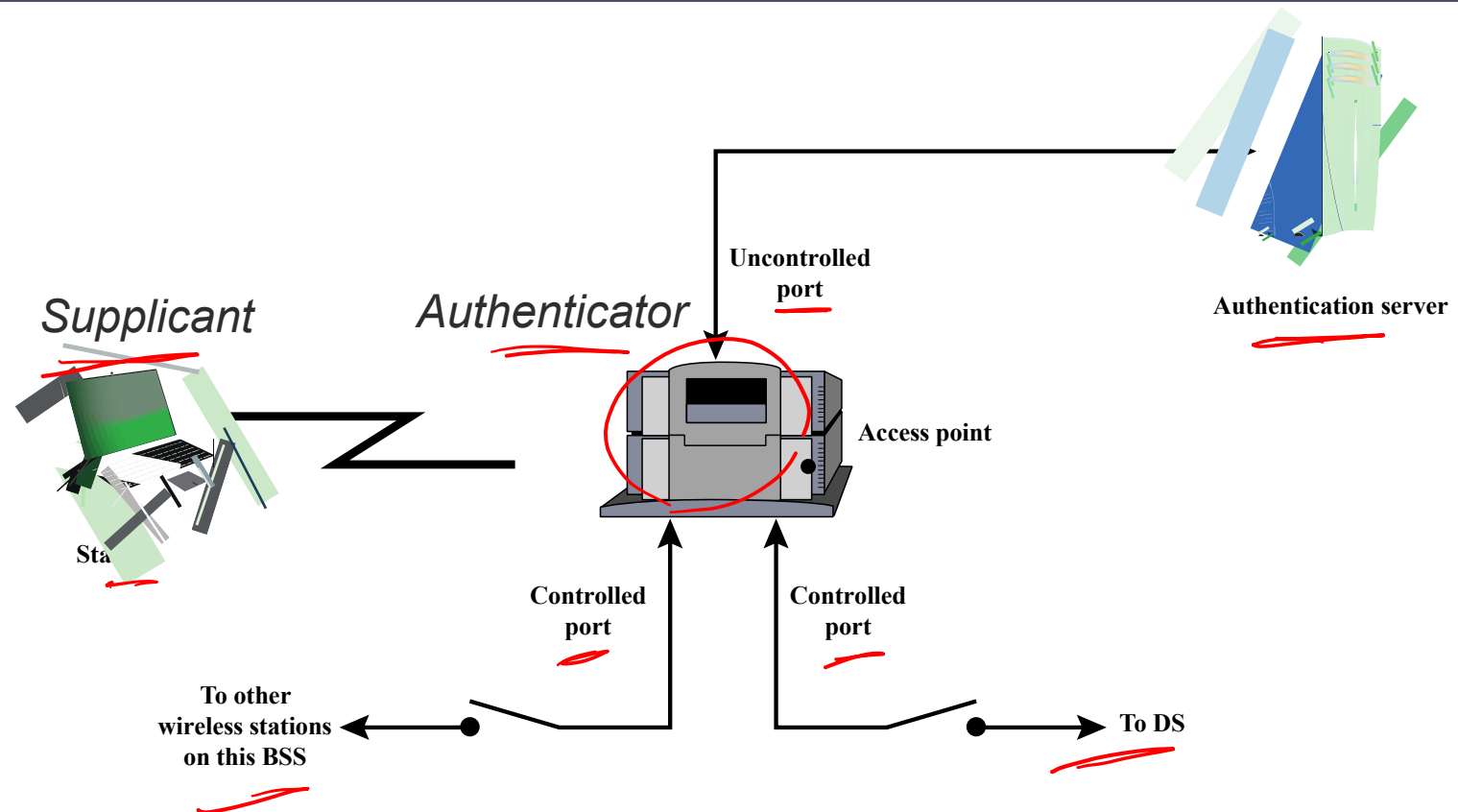


Capability Discovery, Association, and Authentication

Authentication phase consists of three phases:

- **Connect to AS**
 - The STA sends a request to its AS through AP
- **EAP exchange**
 - Authenticates the STA and AS to each other
- **Secure key delivery**
 - On success, the AS generates a Master Session Key and sends it to the STA

802.1X Access Control

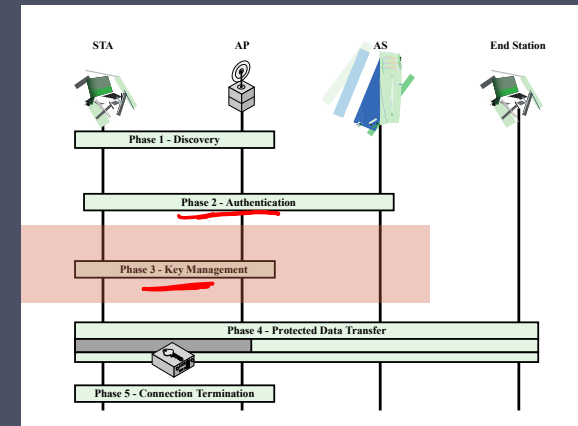


IEEE 802.11i

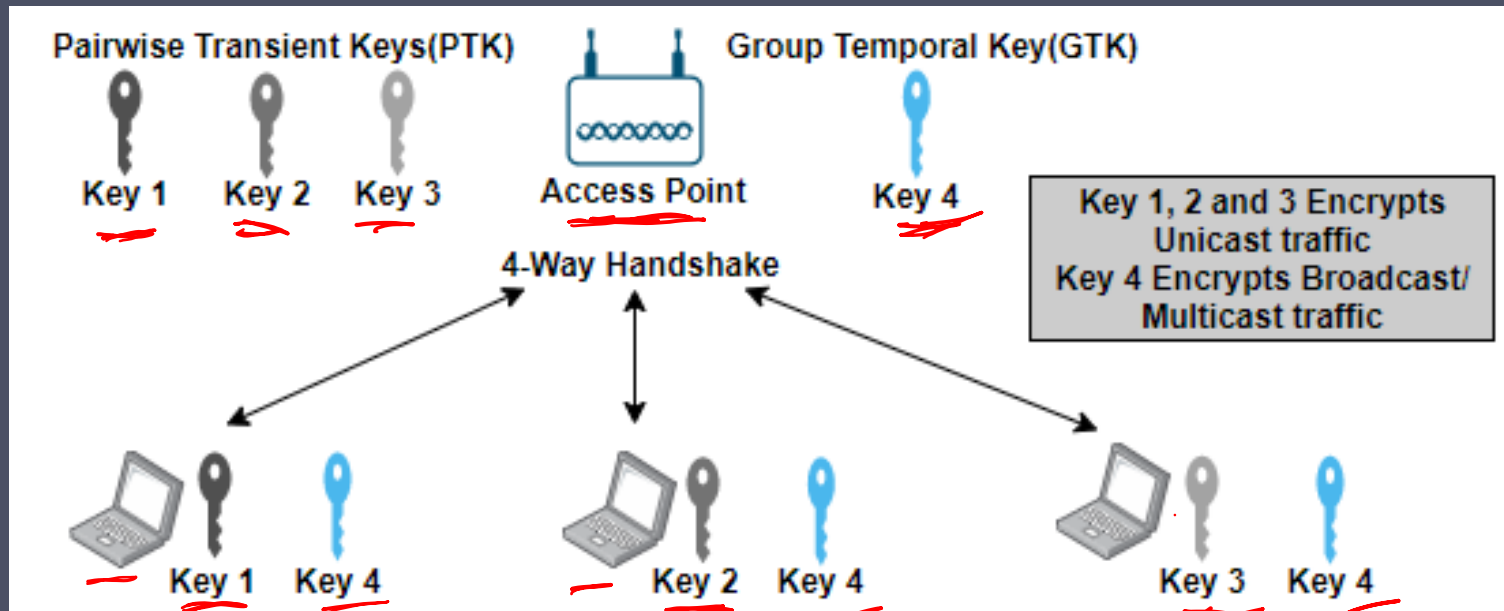
Key Hierarchies



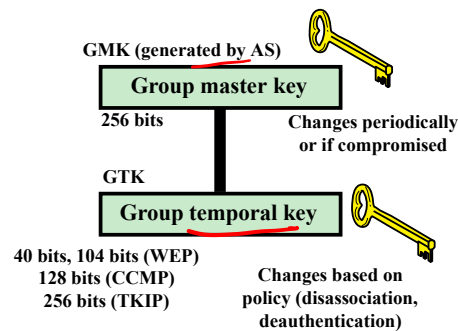
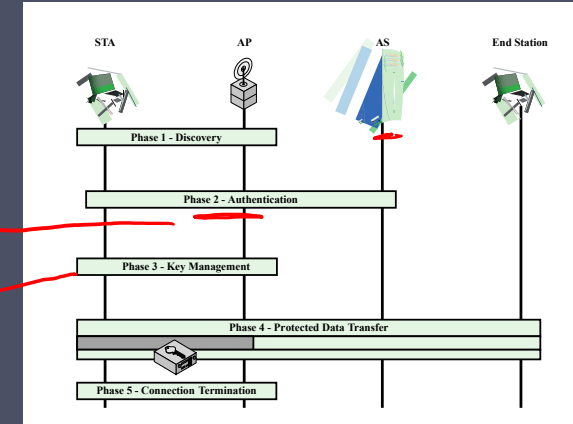
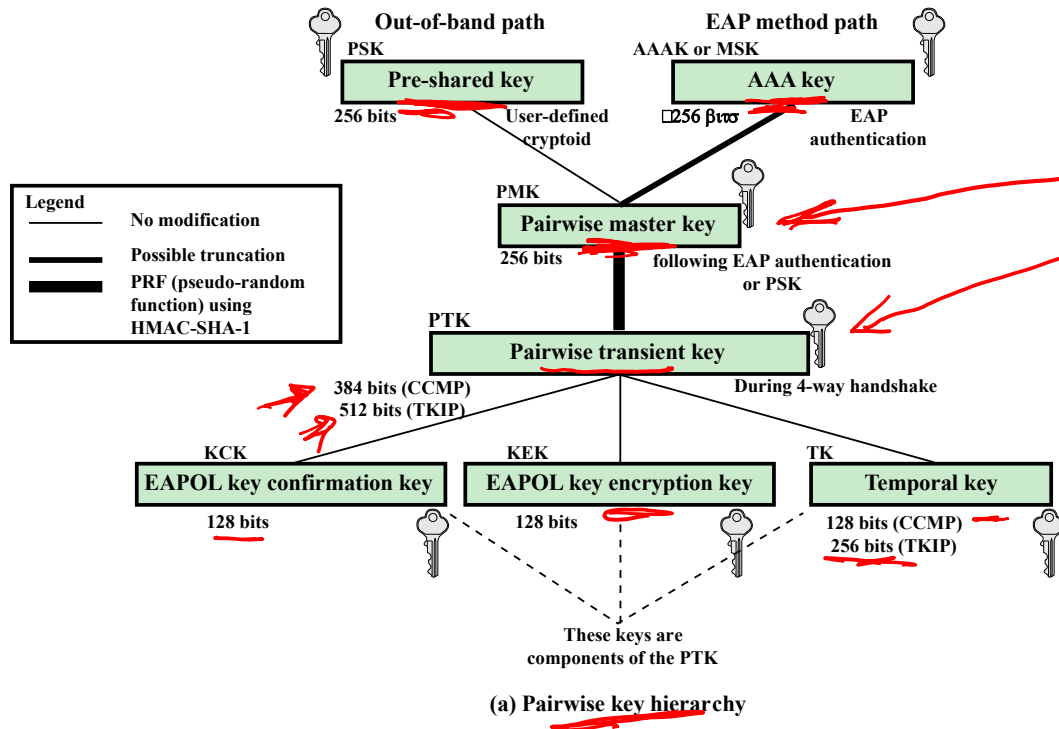
IEEE 802.11i Key Hierarchies



Pairwise keys and Group keys



IEEE 802.11i Key Hierarchies



(b) Group key hierarchy

Abbreviation	Name	Description / Purpose	Size (bits)	Type
AAA Key	Authentication, Accounting, and Authorization Key	Used to derive the PMK. Used with the IEEE 802.1X authentication and key management approach. Same as MMSK.	≥ 256	Key generation key, root key
PSK	Pre-Shared Key	Becomes the PMK in pre-shared key environments.	256	Key generation key, root key
PMK	Pairwise Master Key	Used with other inputs to derive the PTK.	256	Key generation key
GMK	Group Master Key	Used with other inputs to derive the GTK.	128	Key generation key
PTK	Pair-wise Transient Key	Derived from the PMK. Comprises the EAPOL-KCK, EAPOL-KEK, and TK and (for TKIP) the MIC key.	512 (TKIP) 384 (CCMP)	Composite key
TK	Temporal Key	Used with TKIP or CCMP to provide confidentiality and integrity protection for unicast user traffic.	256 (TKIP) 128 (CCMP)	Traffic key
GTK	Group Temporal Key	Derived from the GMK. Used to provide confidentiality and integrity protection for multicast/broadcast user traffic.	256 (TKIP) 128 (CCMP) 40, 104 (WEP)	Traffic key
MIC Key	Message Integrity Code Key	Used by TKIP's Michael MIC to provide integrity protection of messages.	64	Message integrity key
EAPOL-KCK	EAPOL-Key Confirmation Key	Used to provide integrity protection for key material distributed during the 4-Way Handshake.	128	Message integrity key
EAPOL-KEK	EAPOL-Key Encryption Key	Used to ensure the confidentiality of the GTK and other key material in the 4-Way Handshake.	128	Traffic key / key encryption key
WEP Key	Wired Equivalent Privacy Key	Used with WEP.	40, 104	Traffic key

IEEE 802.11i

Keys for Data Confidentiality and Integrity Protocols

IEEE 802.11i Standard

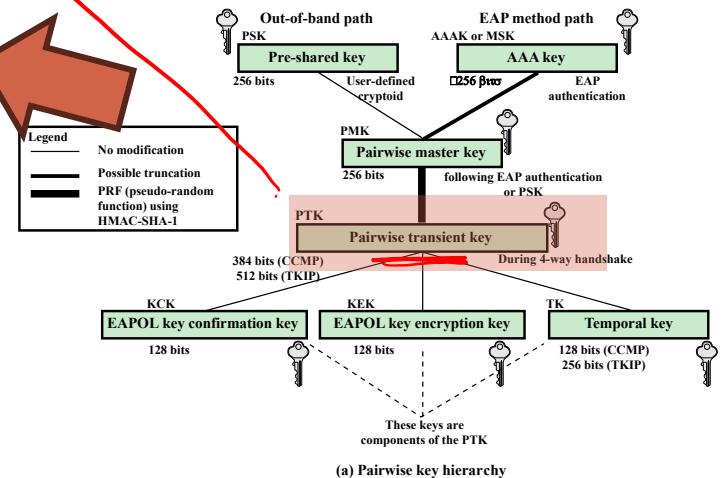
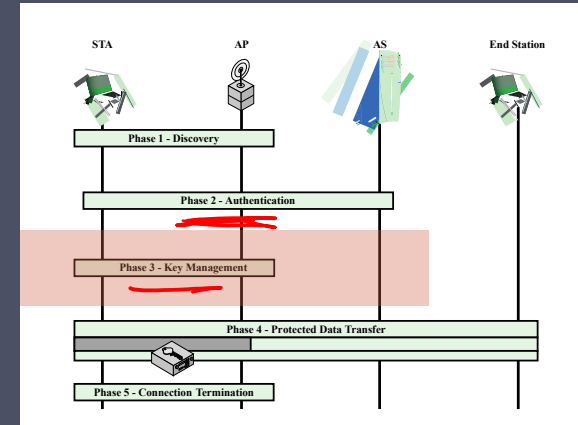
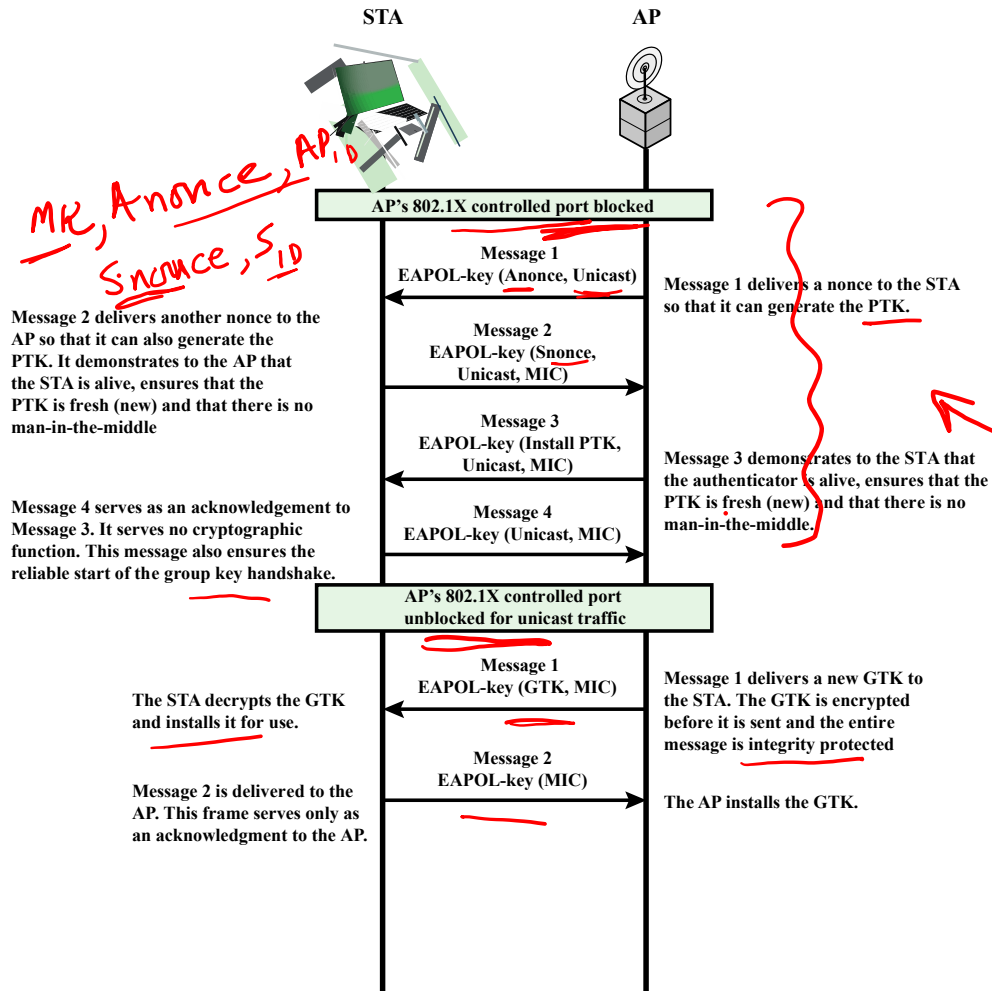
Phases of Operation

→ **4-Way Handshake and Group Key Handshake**

IEEE 802.11i Phases of Operation

Essentially the working WPA2 protocol

4-Way Handshake and Group Key Handshake



About Strength of WPA2

- WPA2 ensures that data sent or received over your wireless network is encrypted, —
 - only people with your network password have access to it.
- WPA2 use AES to replace the more vulnerable TKIP algorithm used in the original WPA protocol.
- WPA2 uses dynamic key encryption, —
 - Regularly changes the key PTK and makes it more difficult to crack.
- However, Pre-Shared Key is derived from passphrase (Wi-Fi password)
 - This is common in Home networks... Enterprise networks use AS and are more safe...
- Weak Wif-Fi passwords can easily be cracked
 - All you need is to capture 4-way handshake exchange messages "airodump-ng"
 - Then use offline dictionary attacks...to crack the password "aircrack-ng"
- That's why we have WPA3 coming as a successor to WPA2...

Buffer Overflow Attacks

A Brief History of Some Buffer Overflow Attacks



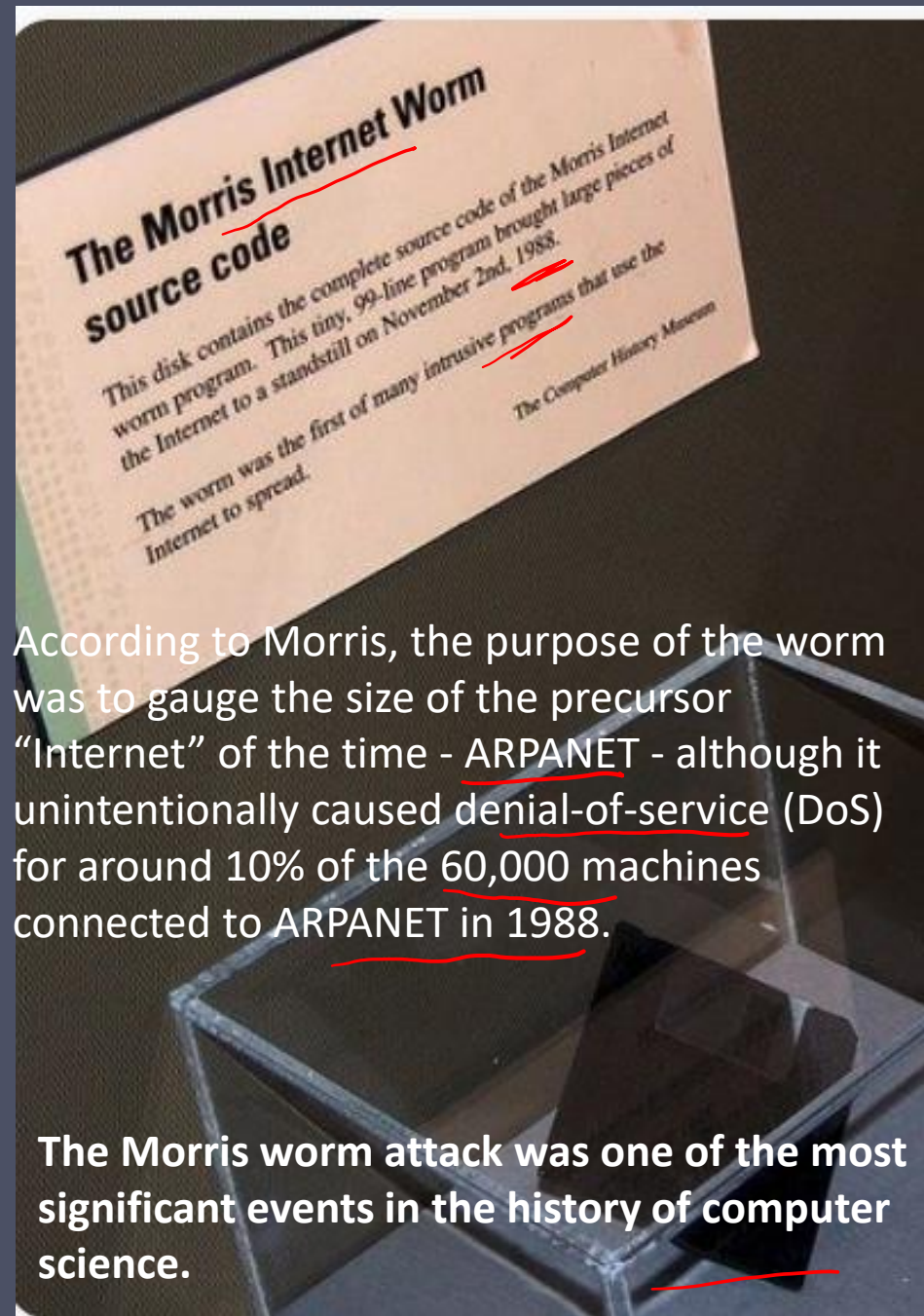
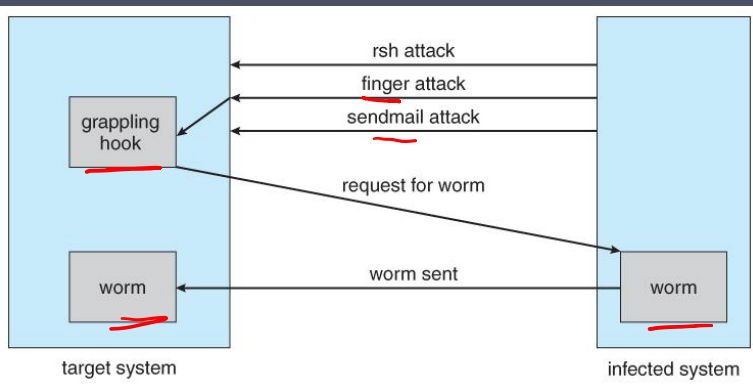
...



Robert Tappan Morris

The Morris worm functioned as a denial-of-service (DoS) attack.

The Morris worm spread by exploiting vulnerabilities in UNIX sendmail (using a backdoor), finger (through buffer overflow), and rsh/rexec (to explore whether it could operate).



According to Morris, the purpose of the worm was to gauge the size of the precursor “Internet” of the time - ARPANET - although it unintentionally caused denial-of-service (DoS) for around 10% of the 60,000 machines connected to ARPANET in 1988.

The Morris worm attack was one of the most significant events in the history of computer science.



Code Red was a computer worm observed on the Internet on July 15, 2001.

This worm targeted Windows-based systems with Microsoft IIS (Internet Information Services for Windows Server) installed exploiting buffer overflow.

The Code Red worm has three phases

- Discovery and propagation... between day 1 and day 19 of any month ✓
- Attack ...between day 20 and day 27 of any month... DDoS attack on White House Web site
- Dormancy...From day 28 to the end of the month

The front page of an infected server might have been changed by the worm to following:

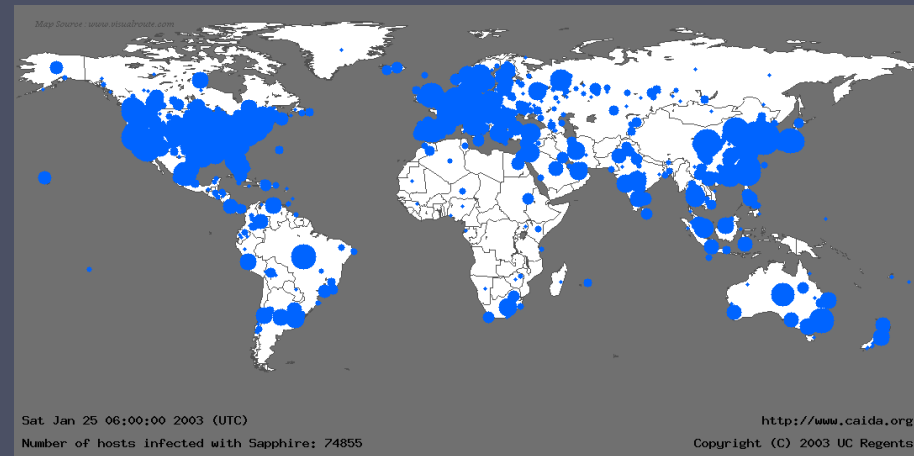


SQL Slammer Virus Attack



SQL Slammer is a 2003 computer worm that caused a denial of service on some Internet hosts and dramatically slowed general Internet traffic and crashing routers all around the world. It spread rapidly, infecting most of its 75,000 victims within 10 minutes.

Slammer exploited a buffer overflow vulnerability in Microsoft SQL server 2000 in order to propagate.





Buffer overflow vulnerability in WhatsApp

On May 14, 2019, a buffer overflow vulnerability was identified in WhatsApp.

The vulnerability allows a remote attacker to execute arbitrary code on the target system.

The vulnerability exists due to a boundary error within the WhatsApp VOIP stack when processing SRTCP packets.

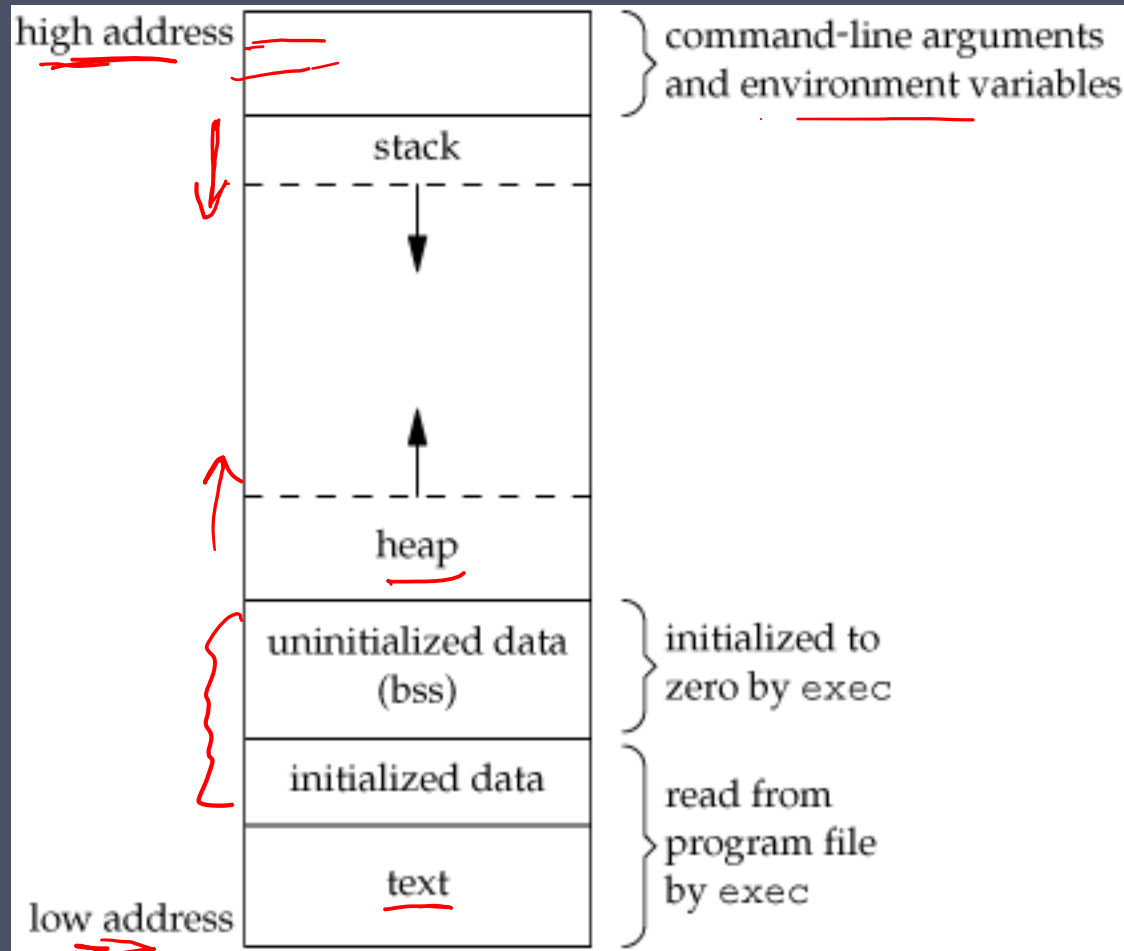
→ A remote attacker can send a series of specially crafted SRTCP packets sent to a target phone number, trigger buffer overflow and execute arbitrary code on the target device.

Successful exploitation of this vulnerability may result in complete compromise of vulnerable system.

✓
https://www.whatsapp.com/security/advisories/archive?lang=en_US

Program Memory Layout and Stack Buffer

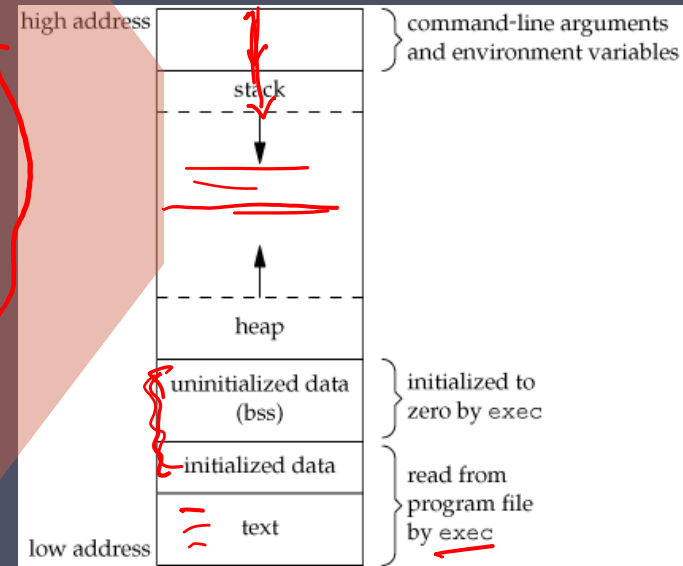
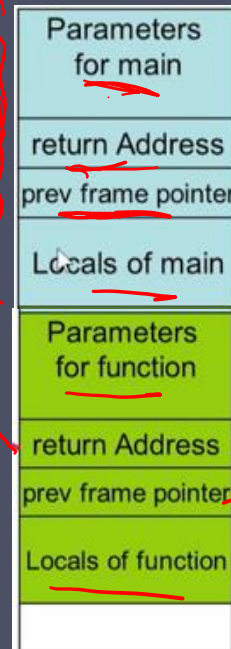
Memory Layout of a C Program



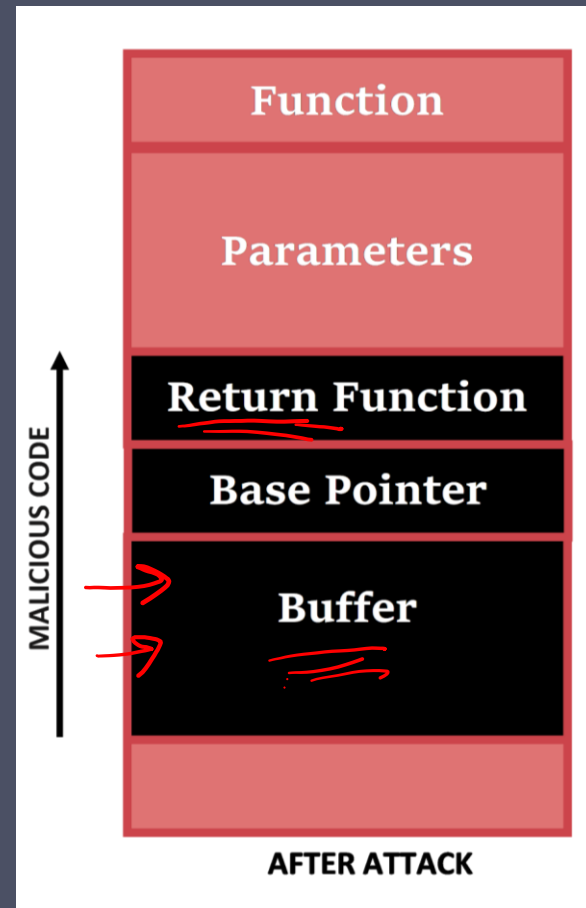
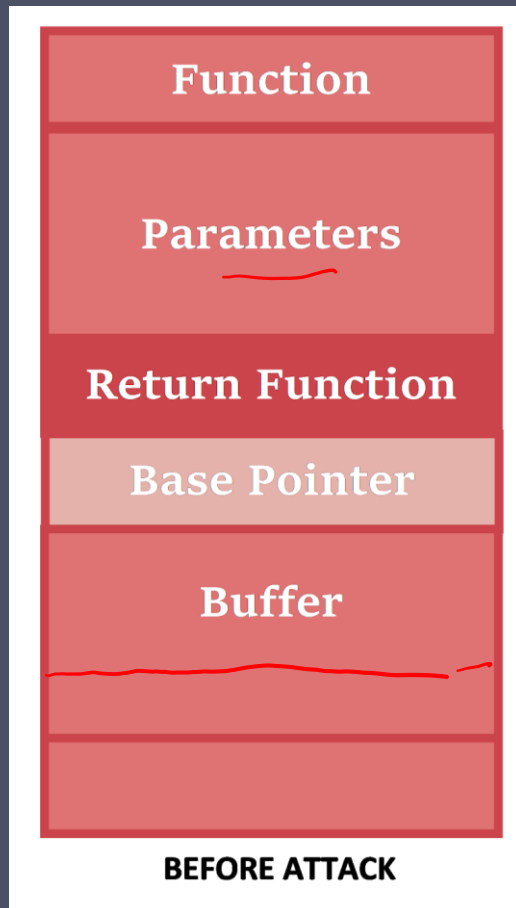
Stack in Program

(when functions are executing)

```
void function(int a, int b, int c){  
    char buffer1[5];  
    char buffer2[10];  
}  
  
int main(int argc, char **argv){  
    function(1,2,3);  
}
```



Buffer Overflow Attack



Buffer Overflow Attack

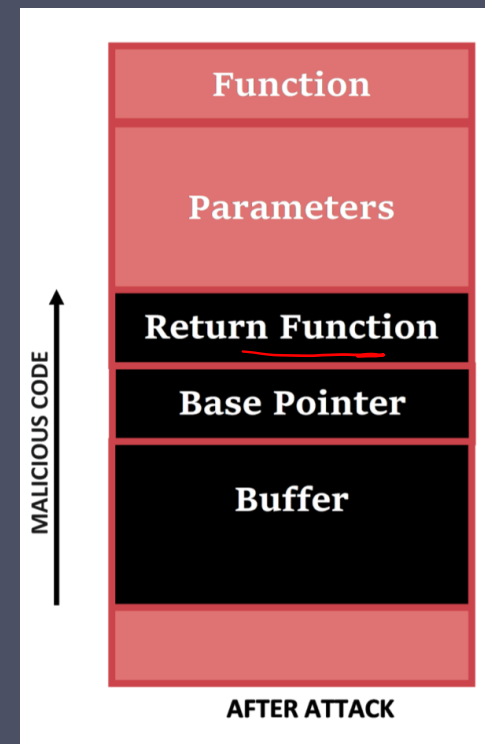
Buffer Overflow Attack

- A very common attack mechanism
 - First widely used by the Morris Worm in 1988
- Prevention techniques known
- Still of major concern
 - Legacy of buggy code in widely deployed operating systems and applications
 - Continued careless programming practices by programmers

Buffer Overflow

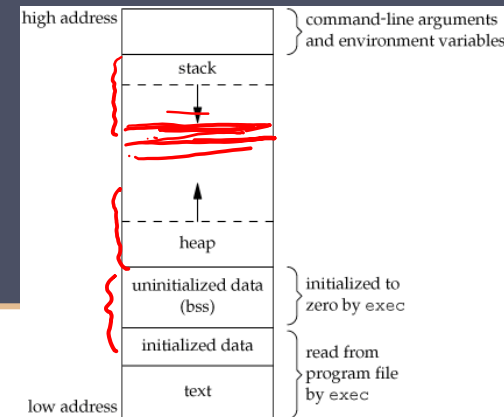
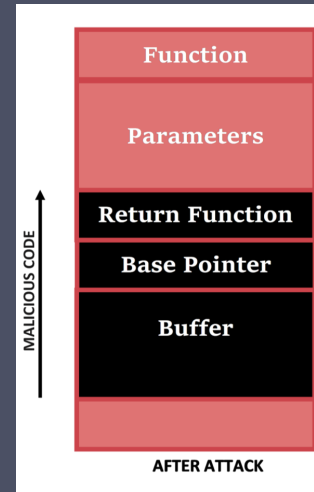
A buffer overflow, also known as a buffer overrun, is defined in the NIST *Glossary of Key Information Security Terms* as follows:

“A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.”



Buffer Overflow Basics

- Programming error when a process attempts to store data beyond the limits of a fixed-sized buffer
- Overwrites adjacent memory locations
 - Locations could hold other program variables, parameters, or program control flow data
- Buffer could be located on the stack, in the heap, or in the data section of the process



Consequences:

- Corruption of program data
- Unexpected transfer of control
- Memory access violations
- Execution of code chosen by attacker

Buffer Overflow Attacks

- To exploit a buffer overflow an attacker needs:
 - To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker's control
 - To understand how that buffer is stored in memory and determine potential for corruption
- Identifying vulnerable programs can be done by:
 - Inspection of program source
 - Tracing the execution of programs as they process oversized input
 - Using tools such as fuzzing to automatically identify potentially vulnerable programs

[Buffer Overflows Made Easy - YouTube](#)

By The Cyber Mentor

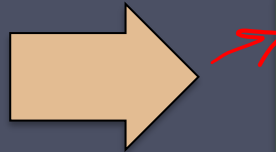
Role of Programming Language in Handling Buffer Overflow Attacks

Programming Language

- At the machine level data manipulated by machine instructions executed by the computer processor are stored in either the processor's registers or in memory
- Assembly language programmer is responsible for the correct interpretation of any saved data value

Modern high-level languages have a strong notion of type and valid operations

- Not vulnerable to buffer overflows
- Does incur overhead, some limits on use



C and related languages have high-level control structures, but allow direct access to memory

- Hence are vulnerable to buffer overflow
- Have a large legacy of widely used, unsafe, and hence vulnerable code

Basic Buffer Overflow Example

```
int main(int argc, char *argv[]) {  
    int valid = FALSE;  
    char str1[8];  
    char str2[8];  
  
    next_tag(str1);  
    gets(str2);  
    if (strncmp(str1, str2, 8) == 0)  
        valid = TRUE;  
    printf("buffer1: str1(%s), str2(%s), valid(%d)\n", str1, str2, valid);  
}
```

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c  
$ ./buffer1  
START  
buffer1: str1(START), str2(START), valid(1)  
$ ./buffer1  
EVILINPUTVALUE  
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)  
$ ./buffer1  
BADINPUTBADINPUT  
buffer1: str1(BADINPUT), str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs

Parameters
for main

return Address
prev frame pointer

Locals of main

Memory
Address

Before
gets(str2)

After
gets(str2)

Contains
Value of

....		
<u>bffffbf4</u>	34fcffbf 4 ...	34fcffbf 3 ...
bffffbf0	01000000	01000000
....		
bffffbec	c6bd0340 ... @	c6bd0340 ... @
bffffbe8	08fcffbf	08fcffbf
....		
bffffbe4	00000000	01000000
....		
bffffbe0	80640140 . d . @	00640140 . d . @
bffffbdc	54001540 T .. @	4e505554 N P U T
bffffbd8	53544152 S T A R	42414449 B A D I
bffffbd4	00850408	4e505554 N P U T
....		
bffffbd0	30561540 0 V . @	42414449 B A D I
....		

argv

argc

return addr

old base ptr

valid

str1[4-7]

str1[0-3]

str2[4-7]

str2[0-3]

Some Common Unsafe C Standard Library Routines

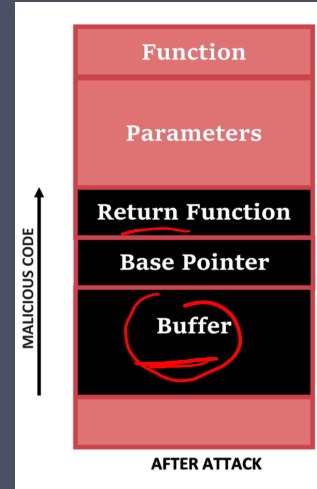
<code>gets(char *str)</code>	read line from standard input into str
<code>sprintf(char *str, char *format, ...)</code>	create str according to supplied format and variables
<code>strcat(char *dest, char *src)</code>	append contents of string src to string dest
<code>strcpy(char *dest, char *src)</code>	copy contents of string src to string dest
<code>vsprintf(char *str, char *fmt, va_list ap)</code>	create str according to supplied format and variables

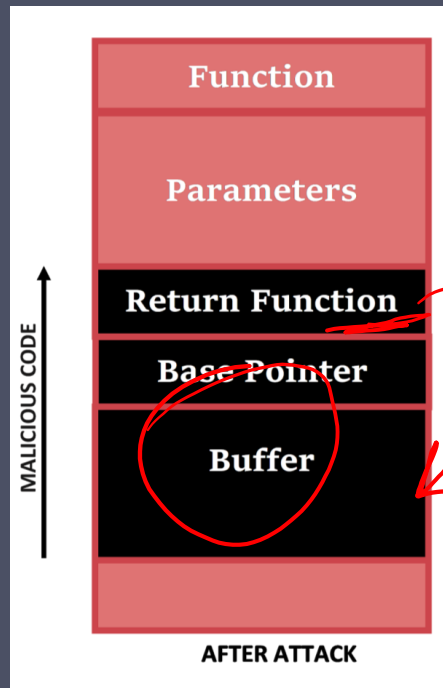
Demo in Visual Studio...

Shellcode

Shellcode

- Code supplied by attacker
 - Often saved in buffer being overflowed
 - Traditionally transferred control to a user command-line interpreter (shell)
- Machine code
 - Specific to processor and operating system
 - Traditionally needed good assembly language skills to create
 - More recently a number of sites and tools have been developed that automate this process
- Metasploit Project
 - Provides useful information to people who perform penetration, IDS signature development, and exploit research





Example UNIX Shellcode

```
int main(int argc, char *argv[])
{
    char *sh;
    char *args[2];

    sh = "/bin/sh";
    args[0] = sh;
    args[1] = NULL;
    execve(sh, args, NULL);
}
```

(a) Desired shellcode code in C

```

nop
nop          // end of nop sled
jmp  find    // jump to end of code
cont: pop  %esi      // pop address of sh off stack into %esi
xor  %eax,%eax    // zero contents of EAX
mov  %al,0x7(%esi) // copy zero byte to end of string sh (%esi)
lea  (%esi),%ebx   // load address of sh (%esi) into %ebx
mov  %ebx,0x8(%esi) // save address of sh in args[0] (%esi+8)
mov  %eax,0xc(%esi) // copy zero to args[1] (%esi+c)
mov  $0xb,%al     // copy execve syscall number (11) to AL
mov  %esi,%ebx    // copy address of sh (%esi) to %ebx
lea  0x8(%esi),%ecx // copy address of args (%esi+8) to %ecx
lea  0xc(%esi),%edx // copy address of args[1] (%esi+c) to %edx
int  $0x80        // software interrupt to execute syscall
find: call cont    // call cont which saves next address on stack
sh:  .string "/bin/sh" // string constant
args: .long 0      // space used for args array
      .long 0      // args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```

90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 89
46 0c b0 0b 89 f3 8d 4e 08 8d 56 0c cd 80 e8 e1
ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

Stack Overflow Variants

Target program
can be:



A trusted system
utility



Network service
daemon



Commonly used
library code

Shellcode functions

Launch a remote shell when connected to

Create a reverse shell that connects back to the
hacker

Use local exploits that establish a shell

Flush firewall rules that currently block other
attacks

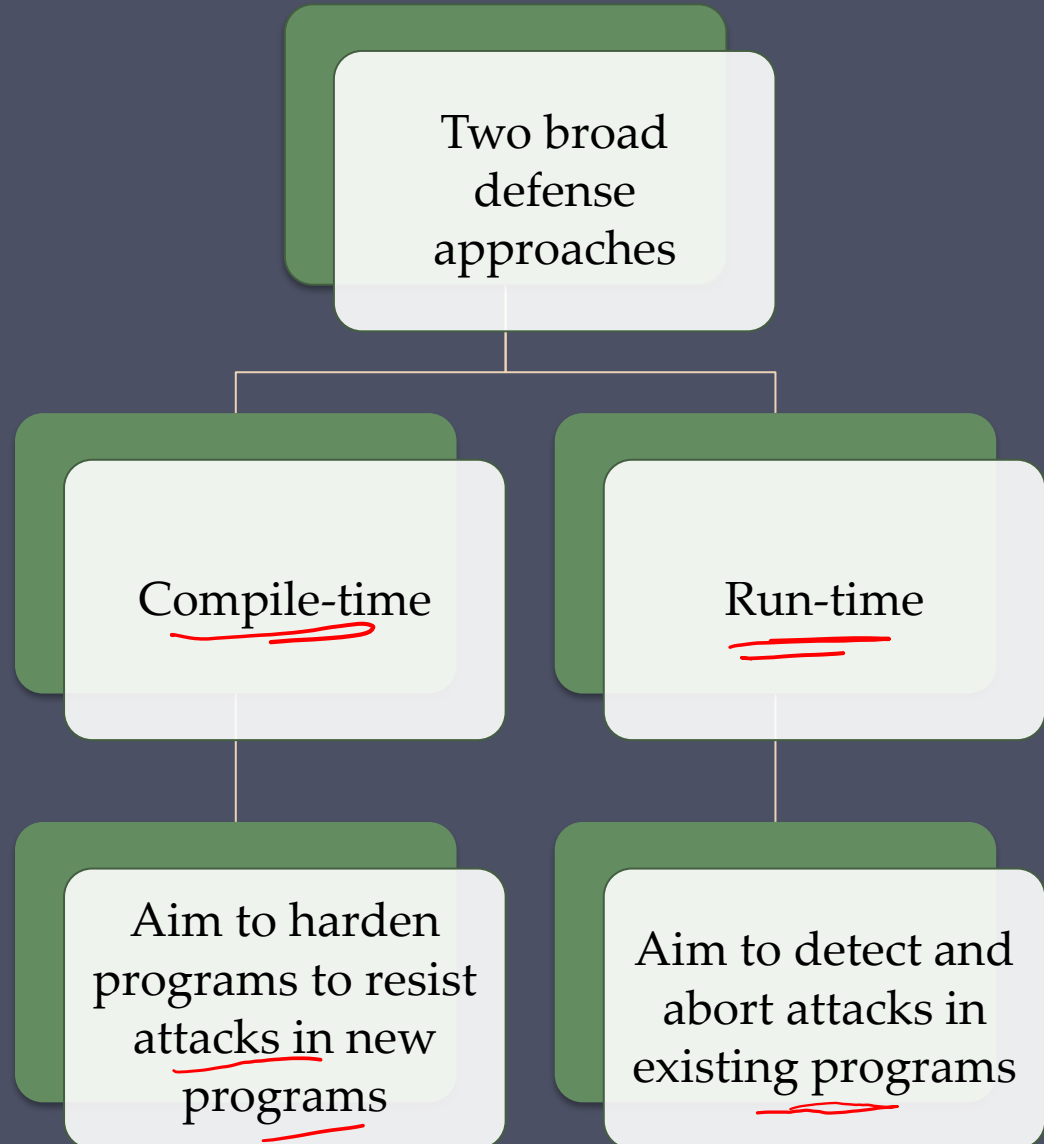


Break out of a chroot (restricted execution)
environment, giving full access to the system

Buffer Overflow Defenses

Buffer Overflow Defenses

- Buffer overflows are widely exploited



Compile-Time Defenses: Programming Language

- Use a modern high-level language
 - Not vulnerable to buffer overflow attacks
 - Compiler enforces range checks and permissible operations on variables

Disadvantages

- Additional code must be executed at run time to impose checks
- Flexibility and safety comes at a cost in resource use
- Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost
- Limits their usefulness in writing code, such as device drivers, that must interact with such resources

Compile-Time Defenses: Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
 - Assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
 - An example of this is the OpenBSD project
- Programmers have audited the existing code base, including the operating system, standard libraries, and common utilities
 - This has resulted in what is widely regarded as one of the safest operating systems in widespread use

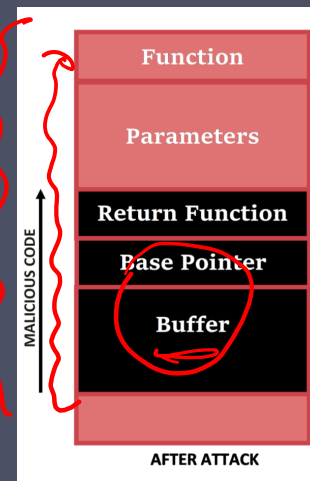
Run-Time Defenses: Executable Address Space Protection

Use virtual memory support to make some regions of memory non-executable

- Requires support from memory management unit (MMU)
- Long existed on SPARC / Solaris systems
- Recent on x86 Linux/Unix/Windows systems

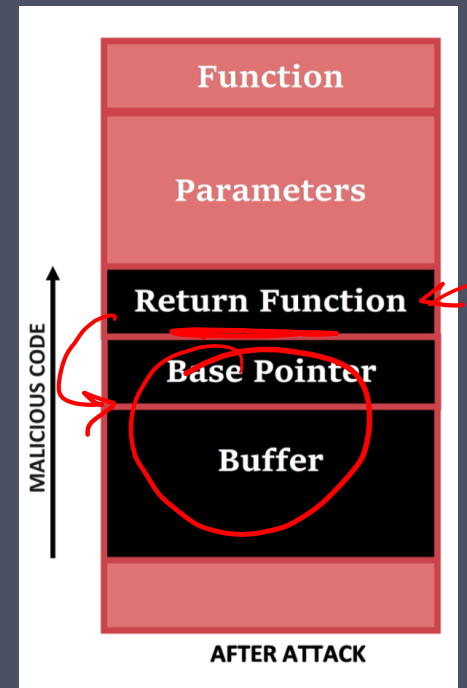
Issues

- Support for executable stack code ?
- Special provisions are needed



Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
 - Stack, heap, global data
 - Using random shift for each process
 - Large address range on modern systems means wasting some has negligible impact
- Randomize location of heap buffers
- Random location of standard library functions



Software Security

CWE/SANS TOP 25 Most Dangerous Software Errors (2011)

Software Error Category: Insecure Interaction Between Components

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Unrestricted Upload of File with Dangerous Type

Cross-Site Request Forgery (CSRF) —

URL Redirection to Untrusted Site ('Open Redirect') —

Software Error Category: Risky Resource Management

Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') —

Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Download of Code Without Integrity Check

Inclusion of Functionality from Untrusted Control Sphere

Use of Potentially Dangerous Function

Incorrect Calculation of Buffer Size

Uncontrolled Format String

Integer Overflow or Wraparound

Software Error Category: Porous Defenses

Missing Authentication for Critical Function

Missing Authorization

Use of Hard-coded Credentials ←

Missing Encryption of Sensitive Data ←

Reliance on Untrusted Inputs in a Security Decision

Execution with Unnecessary Privileges

Incorrect Authorization ←

Incorrect Permission Assignment for Critical Resource ←

Use of a Broken or Risky Cryptographic Algorithm ←

Improper Restriction of Excessive Authentication Attempts ←

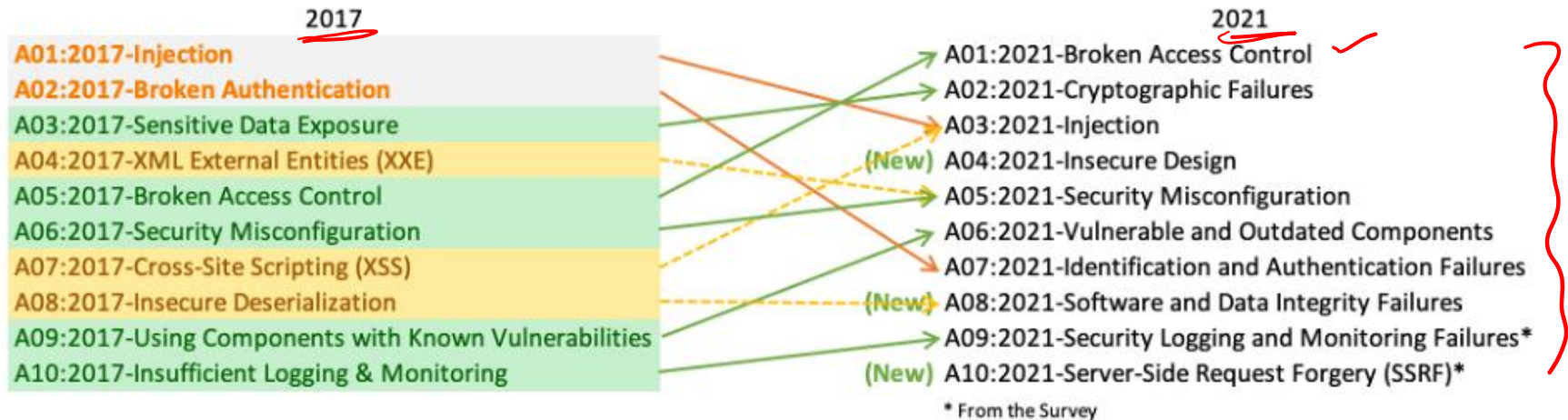
Use of a One-Way Hash without a Salt

OWASP Top 10

Web Application Security Risks

Open Web Application Security Project (OWASP)

There are three new categories, four categories with naming and scoping changes, and some consolidation in the Top 10 for 2021.



<https://owasp.org/www-project-top-ten/> ✓

Security Flaws

- Critical Web application security flaws include five related to insecure software code
 - Unvalidated input
 - Cross-site scripting
 - Buffer overflow
 - Injection flaws
 - Improper error handling
- These flaws occur as a consequence of insufficient checking and validation of data and error codes in programs
- Awareness of these issues is a critical initial step in writing more secure program code
- Emphasis should be placed on the need for software developers to address these known areas of concern

Reducing Software Vulnerabilities

Reducing Software Vulnerabilities

- The NIST report NISTIR 8151 presents a range of approaches to reduce the number of software vulnerabilities
- It recommends:
 - Stopping vulnerabilities before they occur by using improved methods for specifying and building software
 - Finding vulnerabilities before they can be exploited by using better and more efficient testing techniques
 - Reducing the impact of vulnerabilities by building more resilient architectures

Software Security, Quality and Reliability