In-Memory Fuzzing in JAVA

2012.12.17
Xavier ROUSSEL

Summary

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- OWASP definition :

"Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion."

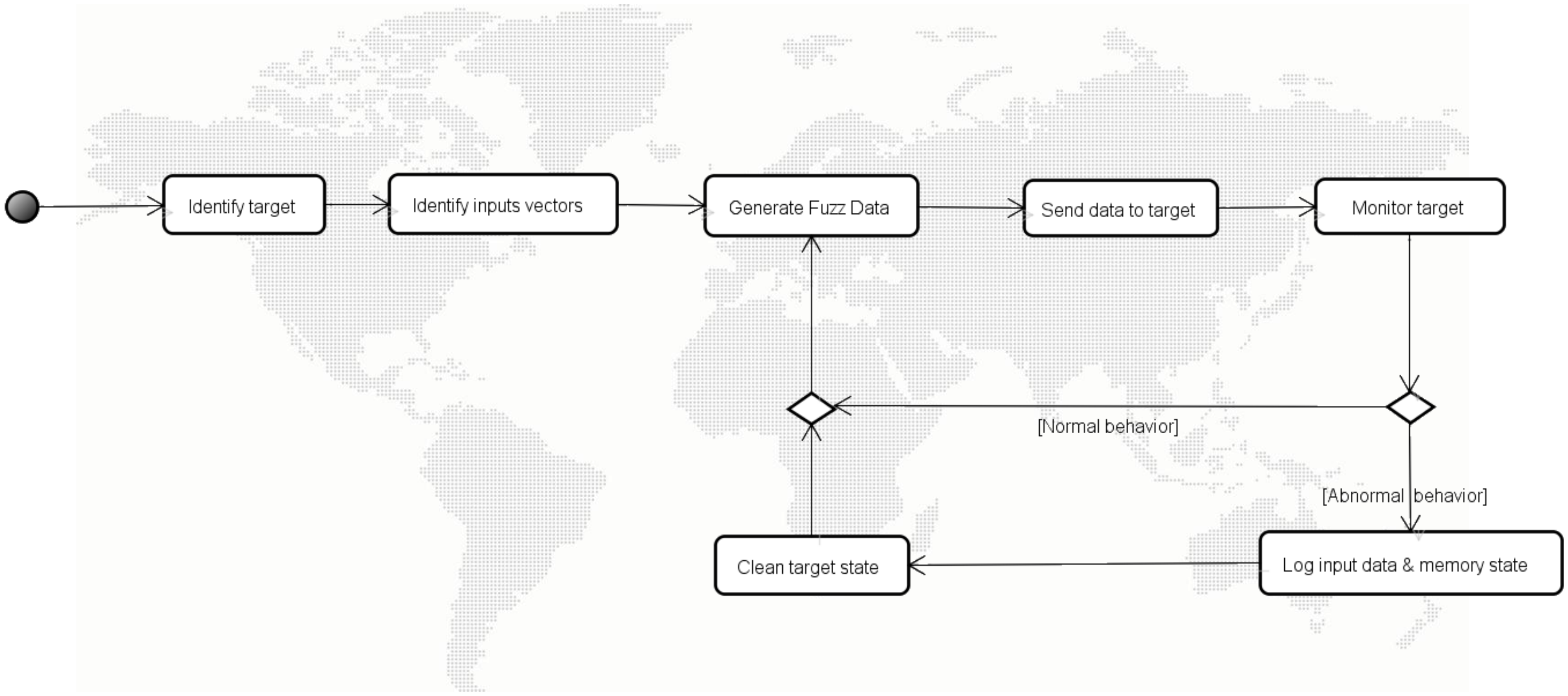
- Alternative to code review mainly used in white box testing.

- Due to automated tests, fuzzing allows us to assess a software against a huge set of test cases in a few time.

- Especially useful to test common applications implementations like FTP server or HTTP server.

I. What is fuzzing?

Fuzzing process



Fuzzing can be used against almost all types of software running on a computer. Preferred targets are privileged applications, remotely accessible applications and file readers.

Example of some commonly targeted applications :

- Server applications (Apache, IIS, etc.)
- Client applications (Internet Explorer, Thunderbird, etc.)
- File readers (Adobe reader, Windows Media Player, etc.)
- Web applications

I. What is fuzzing?

Inputs vectors

Computer security experts commonly use fuzzing to find flaws in software which can lead to system compromise. Attack vectors rely on all components which could be abused to obtain more privileges, mostly:

- Network
- File
- Environment variables
- Execution variables

- **Random-based**

Random-based Fuzzers generate input data for applications in a random way. This type of data generation is very quick to implement but also useless in most cases.

- **Mutation-based**

Mutation-based Fuzzers generate data by analyzing an existing set of data provided by the user and mutating some fields inside these data.

- **Proxy-based**

A proxy-based Fuzzer takes place between a legitimate client and the target server or vice-versa. This architecture allows to capture packets in transition and mutate them before forwarding them to the destination.

- **Specification-based**

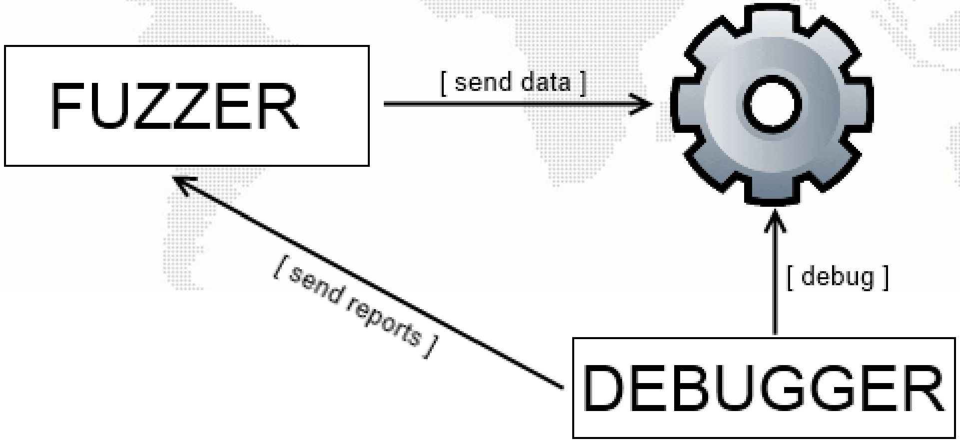
Specification-based Fuzzers generate input data based on specifications of the application. This way, the Fuzzer can test the application very deeply.

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

Target monitoring could be realized in several ways depending on the target application.

- For binary applications, target monitoring could be realized by a debugger to listen for exceptions triggered in the application.

- A web application Fuzzer will analyze page returned by the server to find flaws in the application.



Advantages

- One Fuzzer implementation can be used against all implemented versions of the targeted (e.g. FTP or HTTP).
- A specification-based Fuzzer can quickly audit an application in depth.
- Fuzzing allows software applications testing in black-box.

Drawbacks

- Mutation-based and Random-based Fuzzers are quite quick to implement but in most cases, they can't fuzz the application in depth.
- In the opposite, specification-based Fuzzers can test an application in depth but can be very long to implement.

Summary

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

I. What is Fuzzing?

Introduction
Fuzzing process
Targets
Inputs vectors
Data generation
Target monitoring
Advantages and drawbacks

II. In Memory Fuzzing

Why use in-memory Fuzzing?
Principle
Data injection example
Building in-memory Fuzzer
Creating loop in memory
Advantages and drawbacks

III. DbgHelp4J

Presentation
Key features
Example
Implementing in-memory Fuzzer

IV. Real case study

EasyFTP 1.7.0.11

- As seen before, fuzzing an application require to write a third-party application which allows to launch test cases. That could sometime be difficult if no functions are provided by the target.

- In some case, fuzz testing an application can require a full restart of the latter for each test case. This can lead to very low speed test.

- If an unknown encryption is used by the target application, building an efficient Fuzzer can be quite difficult.

In-memory fuzzing can avoid all these problems by directly injecting fuzz data into memory.

II. In Memory Fuzzing

Principle

- Inject fuzz data directly into memory instead of using the attack vector. Injection can be done by hooking Windows API or a whole function in the process.
- Directly manipulates process memory to clean memory state after each test cases.
- Allow to shortcut data encryption and inject raw data in memory.
- Requires a debugger to place breakpoints and hook key functions.
- Referring to the diagram “Fuzzing process”, in-memory fuzzing operates at the step "Send data to target"

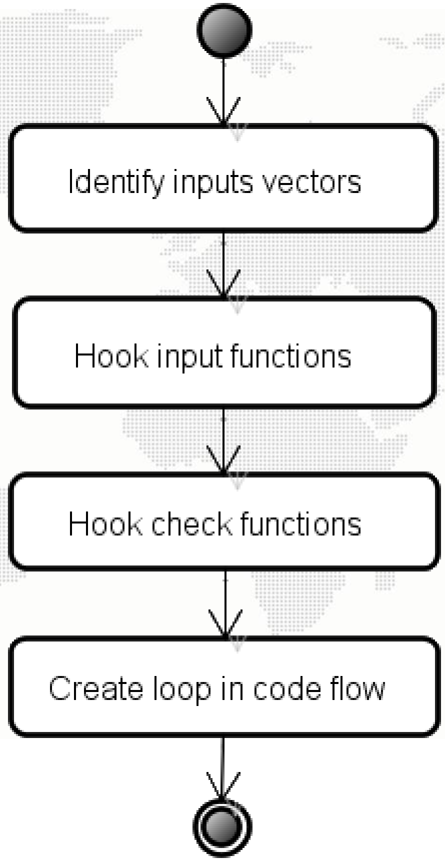
- The code block below use the API “recv”. This API reads data received from the network through a socket connection.

- Hooking this function and replacing the value pointed by the EDX register will allow us to change API's output by our data and thus, to inject our data into the application’s memory.

```
push    0                    ; flags
lea     edx, [esp+18h+buf]
push    1                    ; len
push    edx                  ; buf
push    ebp                  ; s
call    ds:recv
test    eax, eax
jz      short loc_4139B9
```

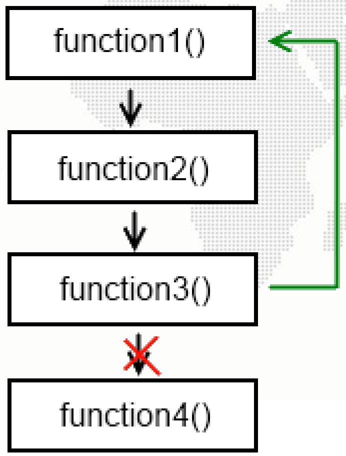
II. In Memory Fuzzing

Building in-memory Fuzzer



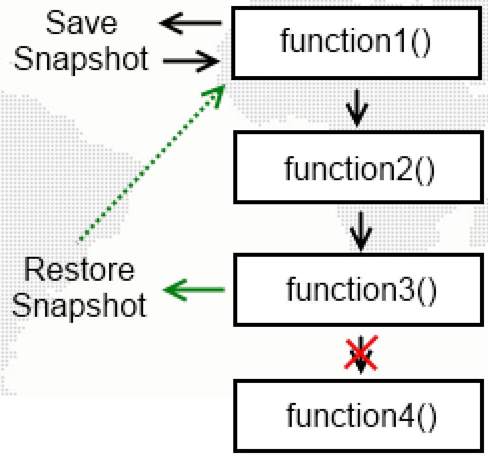
Effective in-memory Fuzzer creates a loop in code flow to restore the memory and allow to launch a new test case. Several ways can be used to create a loop in memory depending on the targeted application.

- Create a loop in memory by manipulating application's code. For example, add a JMP at the end of the function to jump to the beginning of another function previously used in the code flow.



- Obviously, instructions should be adjusted to application's code flow. Stack & heap cleaning might sometime be necessary.

- Another way to create a loop in the code flow is to use memory Snapshots. Memory Snapshot save memory state including threads contexts at the beginning of the loop and restore it at the end of the loop. This way, a loop is virtually created into the code flow and the memory context is restored for each test cases.



HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

Advantages

- Speed : In-memory fuzzing inject data straight into memory and therefore avoid data transfer slowdowns.
- Shortcut : Allows to inject data at desired position and therefore avoid encryption functions or checksum for example.
- Implementation time : avoiding all the different attack vectors, experienced user can build a Fuzzer in a few time.

Drawbacks

- Complexity : build a memory Fuzzer require in-depth analysis of the software and a good knowledge in debugging and assembly language. Forgetting to hook key input functions could make the test ineffective.

Summary

I. What is Fuzzing?

- Introduction
- Fuzzing process
- Targets
- Inputs vectors
- Data generation
- Target monitoring
- Advantages and drawbacks

II. In Memory Fuzzing

- Why use in-memory Fuzzing?
- Principle
- Data injection example
- Building in-memory Fuzzer
- Creating loop in memory
- Advantages and drawbacks

III. DbgHelp4J

- Presentation
- Key features
- Example
- Implementing in-memory Fuzzer

IV. Real case study

- EasyFTP 1.7.0.11

- DbgHelp4J is a JAVA library developed by High-Tech Bridge to debug process in Windows environment.
- It provides all required functions to implements a debug environment and in-memory Fuzzer.
- It provides functionalities to perform static and dynamic binary analysis.
- It permits to perform path analysis.
- It uses the diStorm library to perform binary code analysis.
- It also remains in development.

- Process debugging
- Events listener
- Memory access (Threads, Modules, Process, Windows structures, etc.)
- Read instructions
- Place breakpoints
- Hook functions
- Memory snapshots
- Static / dynamic path analysis

Example – process debug

```java
package org.htbridge.refuzz.core;
import org.htbridge.dbghelp4j.plateform.OsNotFoundException;

public class DebugApp {

    public static void main(String[] args) {

        try {
            WinProcess myProcess = (WinProcess) org.htbridge.dbghelp4j.plateform.System.getSystem().getProcessByName("[PROCESS_NAME]");
            myProcess.attach() ;
            ProcessDebugListener pdl = new ProcessDebugListener() {
                @Override
                public void threadExited(Process p, Thread t) {

                }
                @Override
                public void threadCreated(Process p, Thread t) {

                }
                @Override
                public void rip(Process p) {

                }
                @Override
                public void processExited(Process p) {

                }
                @Override
                public void processCreated(Process p) {

                }
                @Override
                public void outputStringDebug(Process p, StringOutputDebugEvent s) {

                }
                @Override
                public void exceptionThrown(Process p, Thread t, ExceptionDebugEvent e) {

                }
                @Override
                public void dllUnloaded(Process p, ModuleUnloadedDebugEvent m) {

                }
                @Override
                public void dllLoaded(Process p, ModuleLoadedDebugEvent m) {

                }
            };
            myProcess.addDebugListener(pdl) ;
        } catch (ProcessNotFoundException | OsNotFoundException e) {
            e.printStackTrace();
        }

    }
}
```

III. DbgHelp4J

Example – process debug

- Line 18 - WinProcess class owns windows representation of a process
- Line 19 - WinProcess class allows to attach debugger to a process
- Line 20 - ProcessDebugListener sets up debug event listeners
- Line 58 - We attach the debug listeners to the process

- Based on this code, we can easily implement an in-memory Fuzzer using functions from the library.

- We will use memory Snapshots to create the loop.

- Following the process supplied earlier, we first have to identify inputs vectors and hook related functions.

- Here we will use arbitrary address for a “recv” (0x1100) function as for save (0x1000) and restore (0x2000) addresses.

- The first thing to do is to prepare the “recv” hook. It can be achieved by using CallHook class.

```
CallHook ch = new CallHook(myProcess,new Pointer(0x1100)) {
    private Pointer ret ;
    @Override
    public void preCallHook(WinThread t, Pointer address) {
        Memory stack;
        try {
            stack = t.getOwnerProcess().readMemory(new Pointer(((ThreadContextWOW64)t.getContext()).getEsp()), 4*4);
            ret = new Pointer(stack.getInt(4)) ;
        } catch (ReadProcessMemoryException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void postCallHook(WinThread t, Pointer address) {
        ThreadContextWOW64 tcw = ((ThreadContextWOW64)t.getContext()) ;
        tcw.setEax(1) ;
        t.setContext(tcw) ;
        Memory m = new Memory(1) ;
        m.setByte(0, "a".getBytes()[0]) ;
        t.getOwnerProcess().writeMemory(ret,m) ;
        ret = null ;
    }
};
```

- preCallHook function will save pointer address of the string buffer
- postCallHook function will change the value of EAX and insert fuzz value into the string buffer saved previously.

- Next, we have to enable the CallHook for the windows process and put 2 breakpoints to define save and restore addresses.

```
// Prepare recv call hook
myProcess.addCallHook(new Pointer(0x1100), ch) ;
myProcess.enableBreakPoint(ch) ;

// Place save and restore breakpoint
BreakPoint bp = myProcess.addSoftBreakPoint(new Pointer(0x1000)) ;
myProcess.enableBreakPoint(bp) ;
bp = myProcess.addSoftBreakPoint(new Pointer(0x2000)) ;
myProcess.enableBreakPoint(bp) ;
```

- To handle exceptions throws by breakpoints, we have to use the exceptionThrown function from ProcessDebugListener.

```
@Override
public void exceptionThrown(Process p, Thread t, ExceptionDebugEvent e) {
    exceptionHandler(p,t,e) ;
}
```

- The function exceptionHandler will be responsible for saving and restoring memory snapshot.

```
private static void exceptionHandler(Process p, Thread t, ExceptionDebugEvent e) {
    if(Pointer.nativeValue(e.getExceptionAddress()) == Pointer.nativeValue(new Pointer(0x1000)) && ss == null) {
        p.suspend() ;
        ss = p.saveSnapshot() ;
        t.resume() ;
    } else if(Pointer.nativeValue(e.getExceptionAddress()) == Pointer.nativeValue(new Pointer(0x2000))) {
        p.suspend() ;
        p.restoreSnapshot(ss) ;
        t.resume() ;
    }
}
```

- Snapshots will be saved in a global variable.

```
private static Snapshot ss ;
```

- That's it ! Our Fuzzer is now ready. To run the loop, just run a function which reach the save snapshot address.

- This example was an ideal case of in-memory Fuzzer implementation. In real cases, additional functions hooks could be required.

- For example, in many case, there is a select function present before recv function. If select function fail to find the socket (what will certainly happen because the socket connection cannot be kept alive by the Fuzzer) the program will probably take another path and don't reach recv.

- We'll see in the next chapter how to find functions to hook and how to find, save and restore addresses for Snapshots.

Summary

I. What is Fuzzing?

- Introduction
- Fuzzing process
- Targets
- Inputs vectors
- Data generation
- Target monitoring
- Advantages and drawbacks

II. In Memory Fuzzing

- Why use in-memory Fuzzing?
- Principle
- Data injection example
- Building in-memory Fuzzer
- Creating loop in memory
- Advantages and drawbacks

III. DbgHelp4J

- Presentation
- Key features
- Example
- Implementing in-memory Fuzzer

IV. Real case study

- EasyFTP 1.7.0.11

- Now that we know how to implement an in-memory Fuzzer, in this section we will study how to find functions to hook.

- For the Proof of Concept, we'll use an old and well known flaw in EasyFTP 1.7.0.11.

- Following the process supplied earlier, the first thing to do is to identify input vectors.

- Because we work on a FTP server, the main attack vector here is the network.

- The second step is to hook desired input functions. To do this, we must find the address of the these input functions.

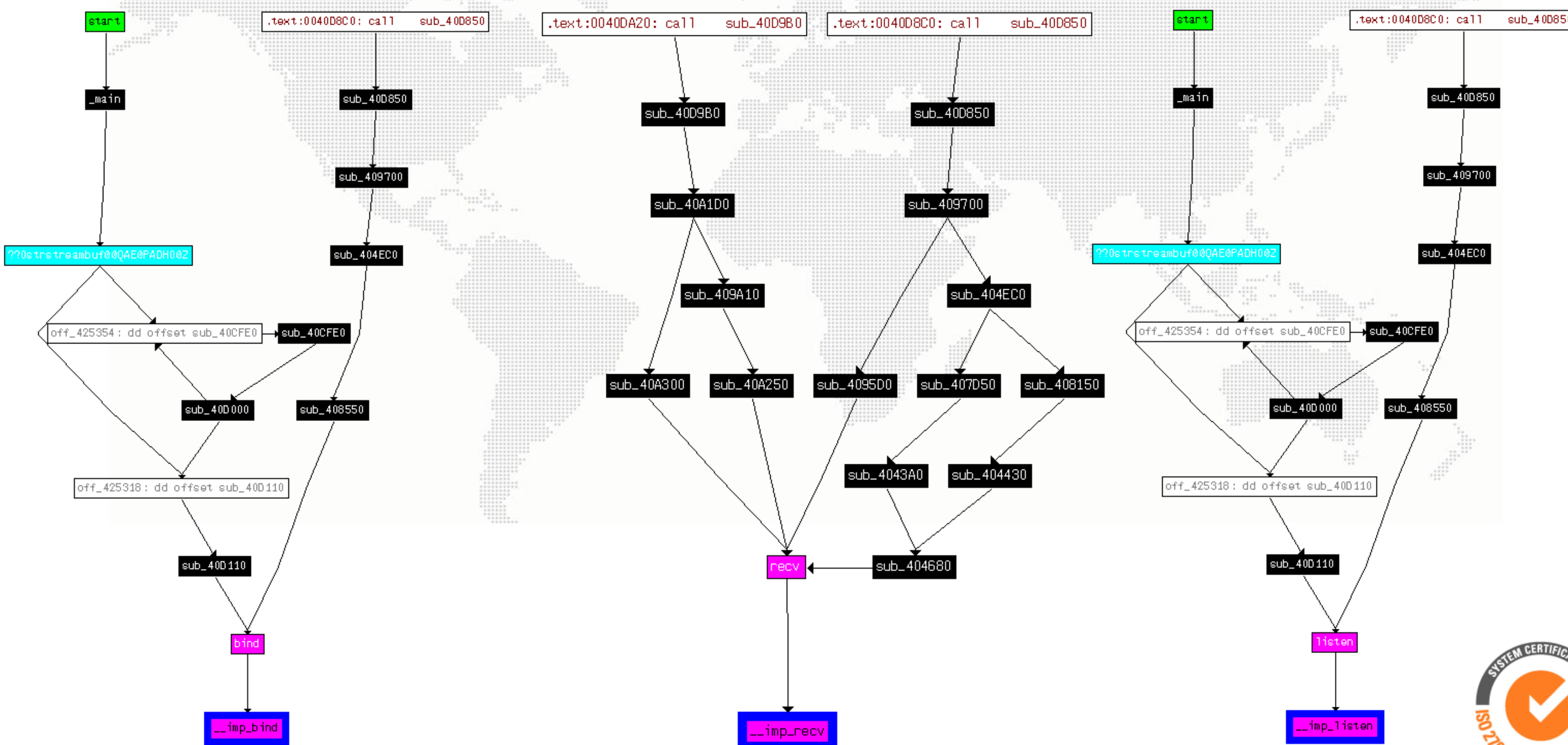
- The best way to proceed is to do a static analysis on the application to find common API used in network communication.

- Here, IDA will be used for static analysis.

- Let's examine the import table of the application :

0042526C	23	socket	WS2_32
00425254	19	send	WS2_32
00425250	18	select	WS2_32
0042524C	16	recv	WS2_32
00425268	15	ntohs	WS2_32
00425230	13	listen	WS2_32
0042523C	10	ioctlsocket	WS2_32
00425244	12	inet_ntoa	WS2_32
00425228	11	inet_addr	WS2_32
00425224	9	htons	WS2_32
00425248	6	getsockname	WS2_32
00425240	5	getpeername	WS2_32
00425264	57	gethostname	WS2_32
00425260	52	gethostbyname	WS2_32
0042521C	4	connect	WS2_32
00425258	3	closesocket	WS2_32
0042522C	2	bind	WS2_32
00425238	1	accept	WS2_32
00425220	151	__WSAFDIsSet	WS2_32
0042525C	115	WSAStartup	WS2_32
00425234	116	WSACleanup	WS2_32

- The most interesting API here is “recv”.
- Other API like “bind”, “select”, “listen” or “accept” could also be useful to help reverse engineering the application.
- Let’s analyze references to the “bind”, “recv” and “listen” API.



IV. Real case study

EasyFTP 1.7.0.11

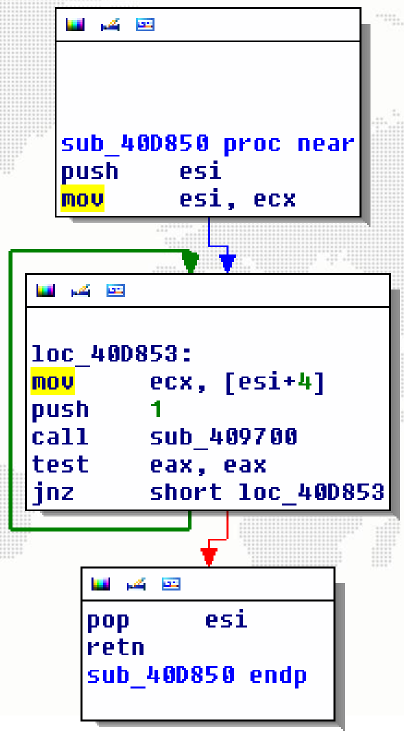
HIGH-TECH BRIDGE® INFORMATION SECURITY SOLUTIONS

- We can easily see that a function 0x40D110 use both “bind” and “listen” API.
- By reversing this function, after “bind” and “listen” calls, an interesting block appears.

```
push    4                   ; unsigned int
call    ??2@YAPAXI@Z        ; operator new(uint)
mov     edx, [ebp+6Ch]
add     esp, 4
mov     esi, eax
push    edi                 ; addrlen
push    edi                 ; addr
push    edx                 ; s
call    accept
push    esi                 ; int
push    edi                 ; dwStackSize
push    offset loc_40D870 ; int
mov     [esi], eax
call    __beginthread
add     esp, 0Ch
jmp     short loc_40D6B6
```

- This block of instructions accepts connections coming from port 21 and launches a thread using function loc_40D870 to handle the connection.

- Inspecting the instruction at loc_40D870, we can find a call to sub_40D850 displayed below.



- Here is the main thread loop. It calls function sub_409700 to receive information and deal with them.

- At this juncture, we have all that we need to create a full loop, but in the graph of the function sub_40D850 presented earlier, we can see that the main thread function is running on a loop. So is it really necessary ?
- Manually implement a loop would be useless because the loop is already present in the code flow.
- Create a loop with memory Snapshots would have the advantage to restore memory to the initial state for each test but this way would prevent the Fuzzer to go deeper into the code as no trace would be kept in memory of previously executed commands.
- Fuzzing the application with the initial code will allow the Fuzzer to go deeper but could also corrupt the memory after several iterations.
- A good alternative here should be to implements memory Snapshots with a counter triggered only after N iterations. Selecting this solution, we should place our save address on the "MOV ESI,ECX" instruction, and restore on "JNZ SHORT LOC_40D853".
- To simplify this example, we will not use a counter here and will restore memory for each test case after connecting.

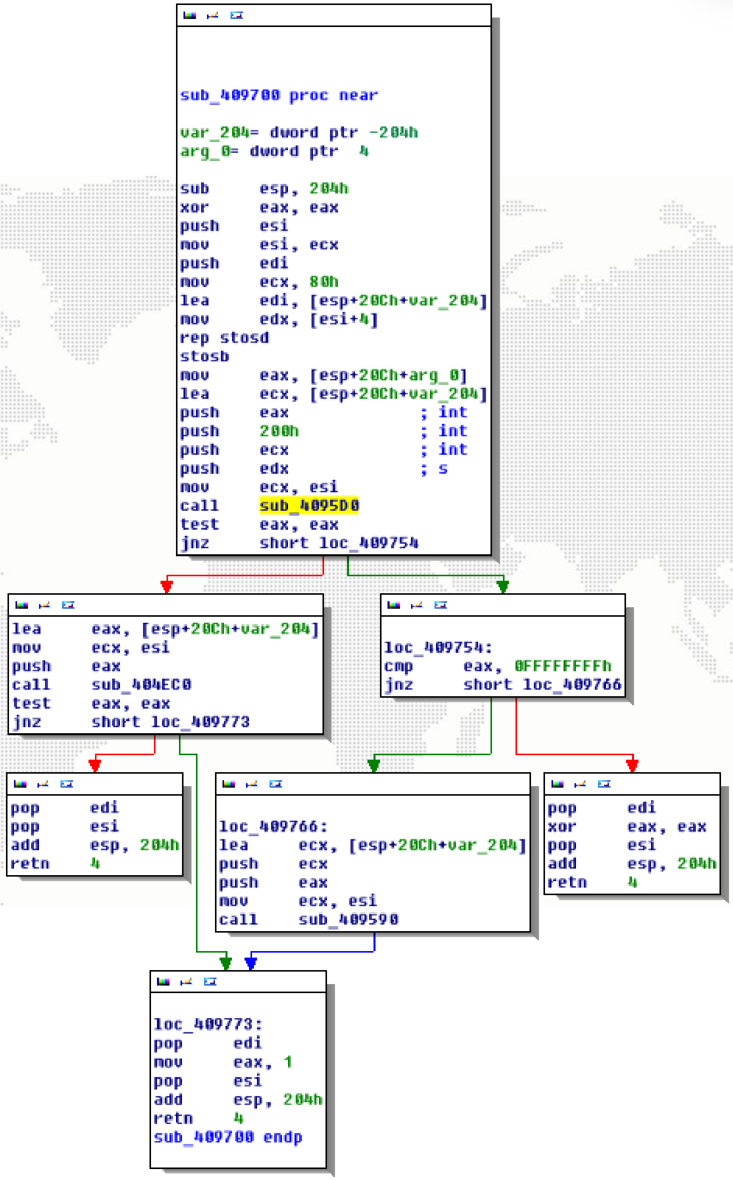
- Having set out our Snapshots addresses, we should now search for the "recv" function to inject our data.

- By inspecting function sub_409700, we find function sub_4095D0 which appears to be the receive function. Here we have 2 solutions. Hook the "recv" call or hook the whole function.

- The second solution appears to be the best one because it allows us to inject data in one block while the "recv" function reads data byte by byte.

IV. Real case study

EasyFTP 1.7.0.11



- Launching the Fuzzer now will reveal another problem. As said earlier, in-memory Fuzzer cannot keep network connection up. Even if we have hooked the "recv" function and so avoid the problem here, the "send" function returns an error and kills the thread.

- So the last thing we have to do is to hook the send function and replace its return value by 1 to entice the application to think the function has terminated correctly.

IV. Real case study

EasyFTP 1.7.0.11 - PoC


```
<terminated> RunnerEasyFTPQuick [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (27 sept. 2012 09:39:13)

[67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68,
Restoring snapshot at native@0x40d85d 102
new test : CWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWDCWD

[67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68, 67, 87, 68,
Restoring snapshot at native@0x40d85d 103
new test : CWD

[67, 87, 68, 32, 32, 32, 32, 32, 0, 0, 0, 0, 0, 0, 13, 10]
Restoring snapshot at native@0x40d85d 104
new test : CWD      iiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii

[67, 87, 68, 32, 32, 32, 32, 32, 0, 0, -20, -20, -20, -20, -20, -20, -20, -20, -20, -20, -20,
Restoring snapshot at native@0x40d85d 105
new test : CWD        0000000000000000000000000000000000000000000000000000000000000000000000

[67, 87, 68, 32, 32, 32, 32, 32, 0, 0, 0, 0, 0, -82, -82, -82, -82, -82, -82, -82, -82, -82,
Restoring snapshot at native@0x40d85d 106
new test : CWD      ::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

[67, 87, 68, 32, 32, 32, 32, 32, 0, 0, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58, 58,
Restoring snapshot at native@0x40d85d 107
new test : CWD   ????????????????????????????????????????????????????????????????????????????

[67, 87, 68, 32, 32, 32, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63,
ACCESS VIOLATION at : native@0x40c67c
Entry : CWD   ????????????????????????????????????????????????????????????????????????????

[67, 87, 68, 32, 32, 32, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63, 63,
Memory dump :

ContextFlags 65599
Dr0 0
Dr1 0
Dr2 0
Dr3 0
Dr6 0
Dr7 0
SegGs 43
SegFs 83
SegEs 43
SegDs 43
Edi 0
Esi 41680036
Ebx 0
Edx 30871824
Ecx 41680036
Eax 1061109567
Ebp 0
Eip 4245116
SegCs 35
EFlags 66054
Esp 41679600
SegSs 43
Stack dump :
native@0x26bc000 [0, 0, 0, 0, 0, 0, 0, 0]
```

- By running the Fuzzer less than 1 minute an “ACCESS VIOLATION” is thrown showing an error in the CWD command handling.

Conclusion

- In this paper, we have discussed about advantages and disadvantages of in-memory fuzzing.

- We have also seen how to build a simple in-memory Fuzzer and analyze the process to place breakpoints and hookpoints.

- In a future paper, we will cover how to harness in-memory fuzzing to help in data generation.

References

- Fuzzing, Brute Force Vulnerability Discovery by Michael Sutton, Adam Greene and Pedram Amini.
- In-Memory Fuzzing on EmbeddedSystems by Andreas Reiter.
- http://resources.infosecinstitute.com/intro-to-fuzzing/
- http://www.ragestorm.net/blogs/
- http://ragestorm.net/distorm/
- https://www.owasp.org/index.php/Fuzzing

Thank you for reading

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

Your questions are always welcome!

xavier.roussel@htbridge.com