

# *PA193 - Secure coding principles and practices*



Dynamic analysis, fuzzing

Petr Švenda

 svenda@fi.muni.cz



@rngsec

CRCS

Centre for Research on  
Cryptography and Security

# Overview

- Lecture:
  - Dynamic analysis of programs for potential bugs
  - Memory analysis
  - Fuzzing (blackbox testing)
  - Tools
- Labs
  - No labs this week because of national holidays
  - Slides/tools available – you can learn anyway (fuzzers)

# DYNAMIC ANALYSIS

# What can dynamic analysis provide

- Dynamic analysis compile and execute tested program
  - real or virtualized processor
- Inputs are supplied and outputs are observed
  - sufficient number of inputs needs to be supplied
  - code coverage should be high
- Memory, function calls and executed operations can be monitored and evaluated
  - invalid access to memory (buffer overflow)
  - memory leak or double free
  - calls to potentially sensitive functions
- <http://www.embedded.com/design/safety-and-security/4419779>

# Techniques used by dynamic analysis

- Debugger (full control over memory read/write, even ops)
- Insert data into program input points (integration tests, fuzzing...)
  - stdin, network, files...
- Insert manipulation proxy between program and library (dll stub, memory)
- Trace of program's external behavior (linux strace)
- Change source code (instrumentation, logging...)
- Change of application binary
- Run in lightweight virtual machine (Valgrind)
- Run in full virtual machine
- Follow propagation of specified values (Taint analysis)
- Mocking (create additional input points into program)
- Restrict programs environment (low memory, limited file descriptors, limited rights...)

# Dynamic analysis tools

- Commercial
  - HP/Fortify, IBM Purify, Veracode, Coverity, Klocwork, Parasoft... (together with static analysis)
- Free
  - GCC gcov tool
  - Valgrind – set of dynamic analysis features
  - Fuzzers
- Most performance analyzers are dynamic analyzers
  - MS Visual Studio→Analyze→Start performance analysis
  - `gcc -Wall -fprofile-arcs -ftest-coverage main.c`
- List of tools for dynamic analysis
  - [https://en.wikipedia.org/wiki/Dynamic\\_program\\_analysis](https://en.wikipedia.org/wiki/Dynamic_program_analysis)

# DEBUGGING SYMBOLS

# Release vs. Debug

- Optimizations applied (compiler-specific settings)
  - gcc `-Ox` (<http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>)
    - `-O0` no optimization (Debug)
    - `-O1 -g / -Og` debug-friendly optimization
    - `-O3` heavy optimization
  - msvc `/Ox /Oi` (<http://msdn.microsoft.com/en-us/library/k1ack8f1.aspx>)
    - MSVS2010: Project properties → C/C++ → optimizations
- Availability of debug information (symbols)
  - gcc `-g`
    - symbols inside binary
  - msvc `/Z7, /Zi`
    - symbols in detached file (`$projectname.pdb`)



# Stripping out debug symbols

- Debug symbols are of great help for an “attacker”
  - key called NSAKey in ADVAPI.dll? (Crypto 1998)
  - <http://www.heise.de/tp/artikel/5/5263/1.html>
- Always strip out debug symbols in released binary
  - check compiler flag
  - Linux: run **file** or **objdump --syms** command (stripped/not stripped)
  - Windows: DependencyWalker

# VALGRIND SUITE

# Valgrind <http://www.valgrind.org/>

- Suite of multiple tools (`valgrind --tool=<toolname>`)
- **Memcheck** - memory management dynamic analysis
  - most commonly used tool (memory leaks)
  - replaces standard C memory allocator with its own implementation and check for memory leaks, corruption (additional guards blocks)...
  - dangling pointers, unclosed file descriptors, uninitialized variables
  - <http://www.valgrind.org/docs/manual/mc-manual.html>
- **Massif** – heap profiler
- **Hellgrind** - detection of concurrent issues
- **Callgrind** – generation of all graphs
- ...

## Valgrind – core options

- Compile with debug symbols
  - `gcc -std=c99 -Wall -g -o program program.c`
  - will allow for more context information in Valgrind report
- Run program with Valgrind attached
  - `valgrind <options> ./program`
  - program cmd line arguments (if any) can be passed
  - `valgrind -v --leak-check=full ./program arg1`
- Trace also into sub-processed
  - `--trace-children=yes`
  - necessary for multi-process / threaded programs
- Display unclosed file descriptors
  - `--track-fds=yes`

## Memcheck – memory leaks

- Detailed report of memory leaks checks
  - `--leak-check=full`
- Memory leaks
  - *Definitely lost*: memory is directly lost (no pointer exists)
  - *Indirectly lost*: only pointers in lost memory points to it
  - *Possibly lost*: address of memory exists somewhere, but might be just randomly correct value (usually real leak)

## Memcheck – uninitialized values

- Detect usage of uninitialized variables
  - `-undef-value-errors=yes` (default)
- Track from where initialized variable comes from
  - `--track-origins=yes`
  - introduces high performance overhead

## Memcheck – invalid reads/writes

- Writes outside allocated memory (buffer overflow)
- Only for memory located on heap!
  - allocated via dynamic allocation (malloc, new)
- Will not detect problems on stack or static (global) variables
  - [https://en.wikipedia.org/wiki/Valgrind#Limitations\\_of\\_Memcheck](https://en.wikipedia.org/wiki/Valgrind#Limitations_of_Memcheck)
- Writes into already de-allocated memory
  - Valgrind tries to defer reallocation of freed memory as long as possible to detect subsequent reads/writes here

# EXAMPLES OF ANALYSIS



```

#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
    int* arrayHeap, unsigned int arrayHeapLen) {
    int Stack[5];

    Static[100] = 0;
    Stack[100] = 0;

    for (int i = 0; i <= 5; i++) Stack [i] = 0;

    int* array = new int[5];
    array[100] = 0;

    arrayStack[100] = 0;
    arrayHeap[100] = 0;

    for (unsigned int i = 0; i <= arrayStackLen; i++) {
        arrayStack[i] = 0;
    }
    for (unsigned int i = 0; i <= arrayHeapLen; i++) {
        arrayHeap[i] = 0;
    }

    return 0;
}

```

```

int main(void) {
    int arrayStack[5];
    int* arrayHeap = new int[5];
    memcheckFailDemo(arrayStack, 5, arrayHeap, 5);
    return 0;
}

```

```

#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
    int* arrayHeap, unsigned int arrayHeapLen) {
    int Stack[5];

    Static[100] = 0; /* Error - Static[100] is out of bounds */
    Stack[100] = 0; /* Error - Stack[100] is out of bounds */

    for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

    int* array = new int[5];
    array[100] = 0; /* Error - array[100] is out of bounds */

    arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
    arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

    for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one */
        arrayStack[i] = 0;
    }
    for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
        arrayHeap[i] = 0;
    }
    /* Problem Memory leak -
return 0;
}

int main(void) {
    int arrayStack[5];
    int* arrayHeap = new int[5];
    memcheckFailDemo(arrayStack, 5, arrayHeap,
return 0;
}

```

## Problems detected – compile time

- `g++ -ansi -Wall -Wextra -g -o test test.cpp`  
– clean compilation

- MSVC (Visual Studio 2012) /W4  
– only one problem detected, `Stack[100] = 0;`

`test.cpp (56)`: error C4789: buffer 'Stack' of size 20 bytes will be overrun; 4 bytes will be written starting at offset 400

- MSVC (Visual Studio 2015) /W4  
– No problem reported

# MSVC /W4

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack,
    int* arrayHeap, unsigned int arrayHeapLen) {
    int Stack[5];

    Static[100] = 0; /* Error - Static[100] is out of bounds */
    Stack[100] = 0; /* Error - Stack[100] is out of bounds */

    for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

    int* array = new int[5];
    array[100] = 0; /* Error - array[100] is out of bounds */

    arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
    arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

    for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one */
        arrayStack[i] = 0;
    }
    for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
        arrayHeap[i] = 0;
    }
    /* Problem Memory leak - array */
    return 0;
}
```

# Visual Studio 2012 & GCC – runtime checks

- Corruption (usually) causes runtime exceptions
  - Stack around variable 'Stack' was corrupted
  - Stack around variable 'arrayStack' was corrupted
- MSVC: /RTC, /GS, /DYNAMICBASE (ASLR) and /NXCOMPAT (DEP)
- GCC: -fstack-protector-all, --no\_execstack (DEP), kernel.randomize\_va\_space=1 (ASLR)
- May preventing successful exploit, but is only last defense

# Cppcheck --enable=all static.cpp



```
[static.cpp:7]: (error) Array 'Static[5]' accessed at index 100, which is out of bounds.  
[static.cpp:8]: (error) Array 'Stack[5]' accessed at index 100, which is out of bounds.  
[static.cpp:10]: (error) Buffer is accessed out of bounds: Stack  
[static.cpp:30] -> [static.cpp:15]: (error) Array 'arrayStack[5]' accessed at  
index 100, which is out of bounds.  
[static.cpp:13]: (error) Array 'array[5]' accessed at index 100, which is out of bounds.  
[static.cpp:25]: (error) Memory leak: array  
[static.cpp:31]: (error) Memory leak: arrayHeap
```

- (Some memory leaks also detected)

# Cppcheck --enable=all file.cpp

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
    int* arrayHeap, unsigned int arrayHeapLen) {
    int Stack[5];

    Static[100] = 0; /* Error - Static[100] is out of bounds */
    Stack[100] = 0; /* Error - Stack[100] is out of bounds */

    for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

    int* array = new int[5];
    array[100] = 0; /* Error - array[100] is out of bounds */

    arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
    arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

    for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one */
        arrayStack[i] = 0;
    }
    for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
        arrayHeap[i] = 0;
    }
    /* Problem Memory leak */
    return 0;
}
/* Not all memory leaks are caught! */
if (1 == 2) delete[] array; /* caught */
if (Stack[0] == 1) delete[] array; /* missed */
if (Stack[0] == 1) delete[] arrayHeap; /*-//-*/
```

## Visual Studio 2012 & PREfast

- Additional two problems detected

- `Static[100] = 0;`

- `for (int i = 0; i <= 5; i++) Stack [i] = 0;`

`test.cpp (55)`: warning : C6200: Index '100' is out of valid index range '0' to '4' **for** non-stack buffer 'int \* Static'.

`test.cpp (58)`: warning : C6201: Index '5' is out of valid index range '0' to '4' **for** possibly stack allocated buffer 'Stack'.

`test.cpp (55)`: warning : C6386: Buffer overrun while writing to 'Static': the writable size is '20' bytes, but '404' bytes might be written.

`test.cpp (62)`: warning : C6386: Buffer overrun while writing to 'array': the writable size is '5\*4' bytes, but '404' bytes might be written.

- `arrayStack` and `arrayHeap` overruns still missed



# Visual Studio 2012 & PRefast

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
    int* arrayHeap, unsigned int arrayHeapLen) {
    int Stack[5];

    Static[100] = 0; /* Error - Static[100] is out of bounds */
    Stack[100] = 0; /* Error - Stack[100] is out of bounds */

    for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

    int* array = new int[5];
    array[100] = 0; /* Error - array[100] is out of bounds */

    arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
    arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

    for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one */
        arrayStack[i] = 0;
    }
    for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
        arrayHeap[i] = 0;
    }
    /* Problem Memory leak - array */
    return 0;
}
```

# Visual Studio 2012 & PREfast & SAL

```
int memcheckFailDemo(  
    _Out_writes_bytes_all_(arrayStackLen) int* arrayStack,  
    unsigned int arrayStackLen,  
    _Out_writes_bytes_all_(arrayHeapLen) int* arrayHeap,  
    unsigned int arrayHeapLen);
```

test.cpp(11): warning : C6200: Index '100' is out of valid index range '0' to '4' **for** non-stack buffer 'int \* Static'.

test.cpp(14): warning : C6201: Index '5' is out of valid index range '0' to '4' **for** possibly stack allocated buffer 'Stack'.

test.cpp(11): warning : C6386: Buffer overrun **while** writing to 'Static': the writable size is '20' bytes, but '404' bytes might be written.

test.cpp(17): warning : C6386: Buffer overrun **while** writing to 'array': the writable size is '5\*4' bytes, but '404' bytes might be written.

test.cpp(23): warning : C6386: Buffer overrun **while** writing to 'arrayStack': the writable size is '\_Old\_2`arrayStackLen' bytes, but '8' bytes might be written.

test.cpp(26): warning : C6386: Buffer overrun **while** writing to 'arrayHeap': the writable size is '\_Old\_2`arrayHeapLen' bytes, but '8' bytes might be written.

# Visual Studio 2012 & PRefast & SAL

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
    int* arrayHeap, unsigned int arrayHeapLen) {
    int Stack[5];

    Static[100] = 0; /* Error - Static[100] is out of bounds */
    Stack[100] = 0; /* Error - Stack[100] is out of bounds */

    for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

    int* array = new int[5];
    array[100] = 0; /* Error - array[100] is out of bounds */

    arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
    arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

    for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one */
        arrayStack[i] = 0;
    }
    for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
        arrayHeap[i] = 0;
    }
    /* Probl
    return for (unsigned int i = 0; i < arrayStackLen + 1; i++) {
        arrayStack[i] = 0;
    }
}
```

# Valgrind --tool=memcheck

```

: valgrind --tool=memcheck ./test
== Invalid write of size 4
==   at 0x4006AB: memcheckFailDemo(int*, unsigned int, int*, unsigned int) (test.cpp:14)
==   by 0x40075D: main (test.cpp:33)
== Address 0x595f230 is not stack'd, malloc'd or (recently) free'd
==
== Invalid write of size 4
==   at 0x4006CB: memcheckFailDemo(int*, unsigned int, int*, unsigned int) (test.cpp:14)
==   by 0x40075D: main (test.cpp:33)
== Address 0x595f1d0 is not stack'd, malloc'd or (recently) free'd
==
== Invalid write of size 4
==   at 0x400710: memcheckFailDemo(int*, unsigned int, int*, unsigned int) (test.cpp:23)
==   by 0x40075D: main (test.cpp:33)
== Address 0x595f054 is 0 bytes after a block of size 20 alloc'd
==   at 0x4C28152: operator new[](unsigned long) (vg_replace_malloc.c:236)
==   by 0x40073F: main (test.cpp:32)
==
== LEAK SUMMARY:
==   definitely lost: 40 bytes in 2 blocks
==
== ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 6 from 6)

```

Invalid write detected  
(array[100] = 0;)

Invalid write detected  
(arrayHeap[100] = 0;)

Invalid write detected  
(arrayHeap[i] = 0;)

Memory leaks detected  
(array, arrayHeap)

# Valgrind --tool=memcheck

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
    int* arrayHeap, unsigned int arrayHeapLen) {
    int Stack[5];

    Static[100] = 0; /* Error - Static[100] is out of bounds */
    Stack[100] = 0; /* Error - Stack[100] is out of bounds */

    for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

    int* array = new int[5];
    array[100] = 0; /* Error - array[100] is out of bounds */

    arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
    arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

    for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one */
        arrayStack[i] = 0;
    }
    for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
        arrayHeap[i] = 0;
    }
    /* Problem Memory leak - array */
    return 0;
}
```

# Valgrind --tool=exp-scheck

```

==15979== Invalid write of size 4
==15979==   at 0x40067C: memcheckFailDemo(int*, unsigned int, int*,
    unsigned int) (test.cpp:11)
==15979==   by 0x40075D: main (test.cpp:33)
==15979==   Address 0x7feffe34 expected vs actual:
==15979==   Expected: stack array "Stack" of size 20 in this frame
==15979==   Actual:   unknown
==15979==   Actual:   is 0 after
==15979== Invalid write of size 4
==15979==   at 0x4006E5: memcheckFailDemo(int*, unsigned int, int*,
    unsigned int) (test.cpp:20)
==15979==   by 0x40075D: main (test.cpp:33)
==15979==   Address 0x7feffe74 expected vs actual:
==15979==   Expected: stack array "arrayStack" of size 20 in frame 1 back from here
==15979==   Actual:   unknown
==15979==   Actual:   is 0 after Expected
==15979==
==15979==
==15979== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 28 from 28)

```

Invalid write detected  
 for (int i = 0; i <= 5; i++) Stack[i] = 0;

Invalid write detected  
 ... arrayStack[i] = 0;

# Valgrind --tool=exp-sgcheck

```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
    int* arrayHeap, unsigned int arrayHeapLen) {
    int Stack[5];

    Static[100] = 0; /* Error - Static[100] is out of bounds */
    Stack[100] = 0; /* Error - Stack[100] is out of bounds */

    for (int i = 0; i <= 5; i++) Stack [i] = 0; /* Error - for Stack[5] */

    int* array = new int[5];
    array[100] = 0; /* Error - array[100] is out of bounds */

    arrayStack[100] = 0; /* Error - arrayStack[100] is out of bounds */
    arrayHeap[100] = 0; /* Error - arrayHeap[100] is out of bounds */

    for (unsigned int i = 0; i <= arrayStackLen; i++) { /* Error - off by one */
        arrayStack[i] = 0;
    }
    for (unsigned int i = 0; i <= arrayHeapLen; i++) { /* Error - off by one */
        arrayHeap[i] = 0;
    }
    /* Problem Memory leak - array */
    return 0;
}
```

# (MSVS 2012) \_CrtDumpMemoryLeaks();

Detected memory leaks!

Dumping objects ->

{155} normal block at 0x00600AD0, 20 bytes **long**.

Data: < > CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD CD

{154} normal block at 0x00600A80, 20 bytes **long**.

Data: < > 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Object dump complete.



# Tools - summary

- *Compilers* (MSVC, GCC) will miss many problems
- *Compiler flags* (/RTC and /GS; **-fstack-protector-all**) flags
  - detect (some) stack based corruptions at runtime
  - additional preventive flags /DYNAMICBASE (ASLR) and /NXCOMPAT (DEP)
- *Valgrind memcheck*
  - will not find stack based problems, only heap corruptions (dynamic allocation)
- *Valgrind exp-sgcheck*
  - will detect stack based problem, but miss first (possibly incorrect) access
- *Cppcheck*
  - detect multiple problems (even memory leaks), but mostly limited to single function
- *PREfast* will find some stack based problems, limited to single function
- *PREfast with SAL* annotations will find additional stack and some heap problems, but not all

# FUZZING (BLACKBOX)



# Beer

HELPING ENGLISH PEOPLE  
HAVE SEX SINCE 1842!



THE ENGLISH PUB  
Austurvöllur

# What is wrong?

Tag 'ff fe' + length of COM section  
 length of comment = length - 2;  
 strlen("hello fuzzy world") == ?

beer.jpg	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f	
00006084	00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f	
00006060	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20	
00006070	20 3c 3f 78 70 61 63 6b 65 74 20 65 6e 64 3d 27	<?xpacket end='
00006084	77 27 3f 3e <b>ff fe 00 14</b> 68 65 6c 6c 6f 20 66 75	w'?>yb..hello fu
00006090	7a 7a 79 20 77 6f 72 6c 64 00 ff db 00 43 00 06	zzy world.yÛ.C..
000060a0	04 05 06 05 04 06 06 05 06 07 07 06 08 0a 10 0a	.....

length of COM section == 00 00  
 length of comment = 0 - 2;  
 -2 == 0xFFFFFFFFFFFFFFFE == ~4GB

beer_fuzzed.jpg*	00 01 02 03 04 05	
00006084	00 01 02 03 04 05	
00006060	20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20	
00006070	20 3c 3f 78 70 61 63 6b 65 74 20 65 6e 64 3d 27	<?xpacket end='
00006084	77 27 3f 3e <b>ff fe 00 00</b> 68 65 6c 6c 6f 20 66 75	w'?>yb..hello fu
00006090	7a 7a 79 20 77 6f 72 6c 64 00 ff db 00 43 00 06	zzy world.yÛ.C..
000060a0	04 05 06	

```
byte* pComment = new byte[MAX_SHORT];
memcpy(pComment, buffer, length);
```

## I love GDI+ vulnerability because...

- Lack of proper ir
- Type signed/uns
- Type overflow
- Buffer overflow
- Heap overflow
- Source code wa
- Huge impact (cc
- Easily exploitabl

**FOUND BY FUZZING** 😊

# INTRO TO FUZZING

## Very simple fuzzer

```
cat /dev/random | ./target_app
```



What do you miss here?



# What is missing?

- Where fuzzing fits in development process? (developer side, CI, SDL)
- What type of bugs fuzzing tends to find?
- Which apps can be fuzzed?
- How to detect that app mishandled fuzzed input (“hit”)? (crash, signal, exception, error...)
- How to react on detected “hit”? (save seed and crashing inputs, bucketing of inputs)
- How to create more meaningful inputs than random bytes? (valid inputs, proxy)
- How to fuzz non-binary inputs? (string patterns, regex, mouse movements...)
- How to fuzz applications without input as files? (http requests, dll injection, ZAP example)
- How to fuzz efficiently? (known problematic values (fuzz vectors))
- How to fuzz files/inputs with defined structure? (grammar, example Peach)
- How to make fuzzer protocol-aware? (Peach example)
- How to fuzz state-full protocols? (proxy like fuzzing)
- How to analyse and react on detected hits?
- Which tools one can use?
- How to detect less visible “hits”? (side-channels)
- What else can we fuzz? (test coverage testing, DDOS resiliency, hardware inputs)







## Fuzzing: key characteristics

1. More or less random modification of inputs
2. Monitoring of target application
3. Huge amount of inputs for target are send
4. Automated and repeatable

# Fuzzing - advantages/disadvantages

- Fuzzing advantages 
  - Very simple design
  - Allow to find bugs missed by human eye
  - Sometimes the only way to test (closed system)
  - Repeatable (crash inputs stored)
- Fuzzing disadvantages 
  - Usually simpler bugs found (low hanging fruit)
  - Increased difficulty to evaluate impact or dangerousity
  - Closed system is often evaluated, black box testing

## What kind of bugs is usually found?

- Memory corruption bugs (buffer overflows...)
- Parser bugs (crash of parser on malformed input)
- Invalid error handling (other than expected error)
- Threading errors (requires sufficient setup)
- Correctness bugs (reference vs. new impl.)

## What kind of bugs are usually missed?

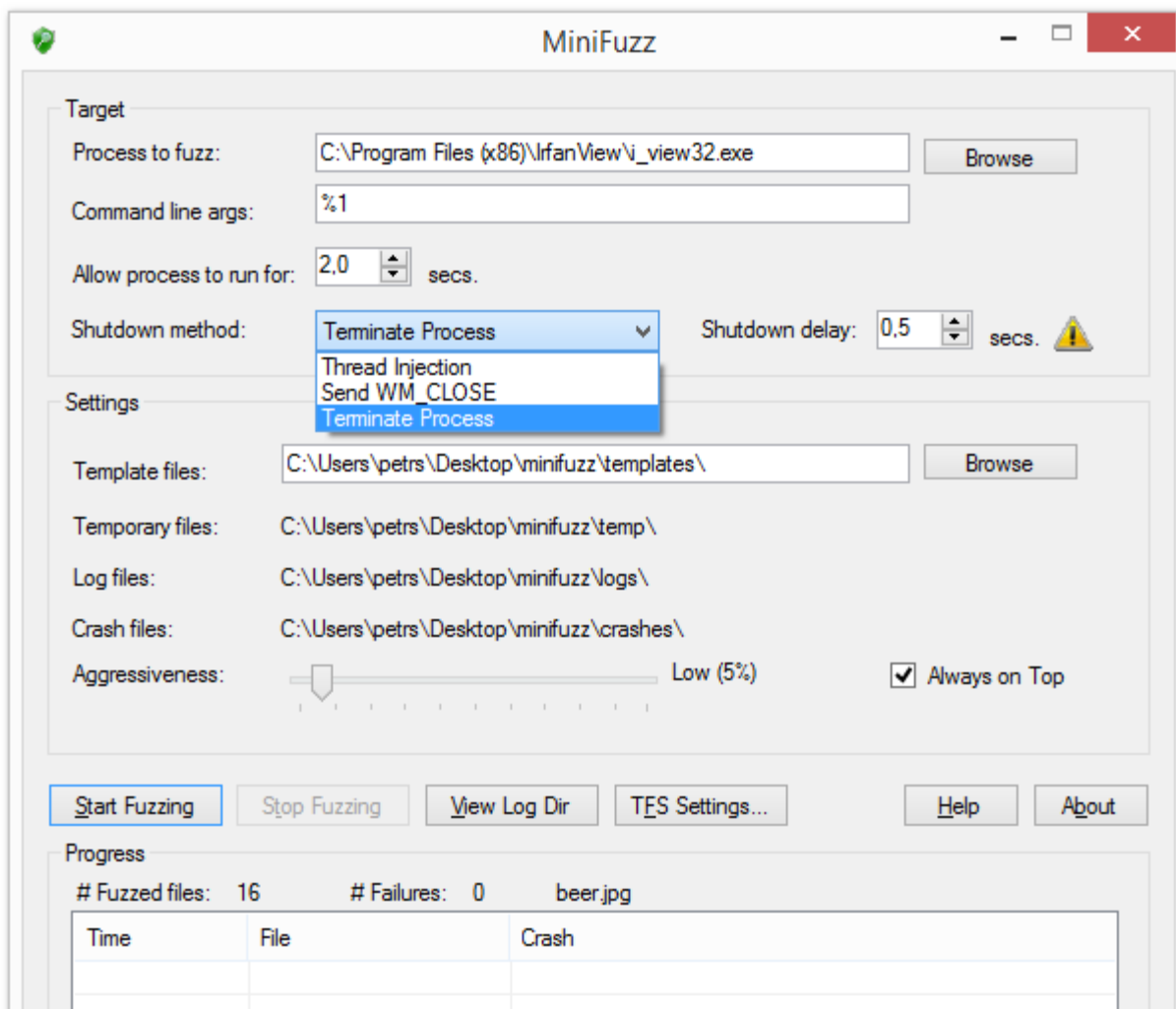
- Bugs after input validation (if not modeled properly)
- High-level / architecture bugs (e.g. weak crypto)
- Usability bugs
- ...

# What kind of applications can be fuzzed?

- Any application/module with an input
  - (sometimes even without inputs, e.g., fault induction)
- Custom (“DIY”) fuzzer
  - Usually full knowledge about target app
  - Kind of randomized integration test (but still repeatable!)
- File fuzzer – input via files
- Network fuzzer – input received via network
- General fuzzing framework
  - Preprepared tools and functions for common tasks (file, packet...)
  - Custom plugins, pre-prepared and custom data models

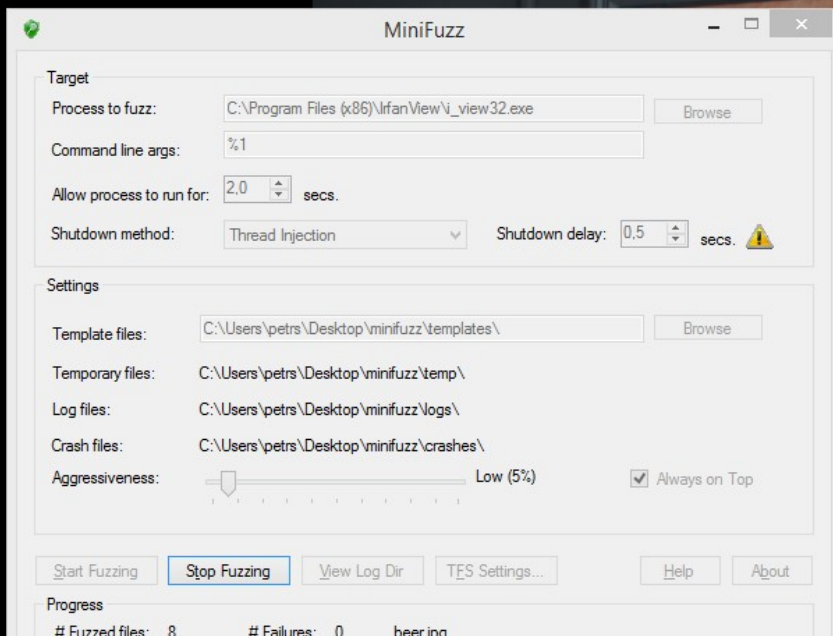


# Microsoft's SDL MiniFuzz File Fuzzer





File Edit Image Options View Help



&lt;?xml version="1.0"?&gt;

&lt;failures&gt;

<failure type="Exception Event:Tid=8504, 0x8000003, unhandled, address=0x7740e34d" datetime="11:21:12 12.

<registers RAX="00000000" RBX="00000000" RCX="7FFF5FC5180A" RDX="00000000" RSI="00000000" RDI="00

<process name="C:\Program Files (x86)\IrfanView\i\_view32.exe" />

<file name="-std=c99 -Wall C:\minifuzz\temp\beer-0rsw9!h2jf.jpg" />

&lt;/failure&gt;

&lt;/failures&gt;





# MiniFuzz: gcc fuzzing

Target

Process to fuzz: C:\MinGW\bin\gcc.exe

Command line args: -std=c99 -Wall %1

Allow process to run for: 2.0 secs.

Shutdown method: Thread Injection Shutdown delay: 0.5 secs.

Settings

Template files: C:\Users\pets\Desktop\minifuzz\templates\

Temporary files: C:\Users\pets\Desktop\minifuzz\temp\

Log files: C:\Users\pets\Desktop\minifuzz\logs\

Crash files: C:\Users\pets\Desktop\minifuzz\crashes\

Aggressiveness:

Progress

# Fuzzed files: 65 # Failures: 1 hello.c

Time	File	Crash
11:21 12.72	gcc.exe	0x80000003 unhandled address=0x7740e34d

```
#include<stdio.h>
int main() {
    printf("Hello Fuzzy World");
    return 0;
}
```

Binary fuzzing of source code???

How to improve test coverage

What if file is not an input?

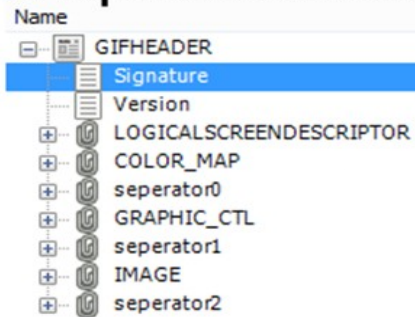




# What kind of inputs and strategy?

- Type of inputs?
  - File, network packets, structure, data model, state(-less)
- What environment setup is necessary?
  - Fuzzing on live system?
  - Multiple entities inside VMs? Networking?
- Isolated vs. cooperating components?
  - We don't like to mock everything
- What tools are readily available?

## 2. Prepare data model



<http://iconarchive.com>,  
<http://awicons.com>,  
<http://www.pelfusion.com>

# MODELLING

# Input preparation

- *Time intensive part of fuzzing (if model !exists yet)*
  1. Fully random data
  2. Random modification of valid input
  3. Modification of valid input with fuzz vectors
  4. Modification of valid input with mutator
  5. Fuzzing via intermediate proxy



## Radamsa fuzzer

- “...easy-to-set-up general purpose shotgun test to expose the easiest cracks...”
  - <https://code.google.com/p/ouspg/wiki/Radamsa>
- Just provide input files, all other settings automatic
  - **cat** file | radamsa > **file.fuzzed**

```
>echo "1 + (2 + (3 + 4))" | radamsa --seed 12 -n 4
1 + (2 + (2 + (3 + 4?))
1 + (2 + (3 +?4))
18446744073709551615 + 4)))
1 + (2 + (3 + 170141183460469231731687303715884105727)))
```

## Fuzzing via intermediate proxy

- Fuzzer modifies valid flow according to data model
- Usually used for fuzzing of state-full protocols
  - Modelling states and interactions would be difficult
  - Target application(s) takes care of states and valid input





# OWASP's ZAP – fuzz strategy settings



Quick Start \* Request Response ← Break Script Console

## Welcome to the OWASP Zed Attack Proxy (ZAP)

ZAP is an easy to use integrated penetration testing tool. Please be aware that you should only attack applications that you own or have been granted permission to test. To quickly test an application, enter its URL below.

URL to attack:

Progress: Not started

For a more in depth test you should explore your target application. If you are using Firefox 24.0 or later you can use the Firefox ZAP Add-on.

Configure your browser:

Or point your browser at:

### Options

- Options
  - Active Scan
  - Active Scan Input Vectors
  - AJAX Spider
  - Anti CSRF Tokens
  - API
  - Applications
  - Authentication (Dependencies)
  - Breakpoints
  - Certificate
  - Check For Updates
  - Connection
  - Database
  - Display
  - Dynamic SSL Certificates
  - Encode/Decode
  - Extensions
  - Forced Browse
  - Fuzzer
  - Global Exclude URIs

### Fuzzer

Default category:

Concurrent scanning threads:

Add custom Fuzz file:

- jbrofuzz / XSS
- jbrofuzz / SQL Injection
- jbrofuzz / URI Exploits
- jbrofuzz / User Agents
- jbrofuzz / Web Server
- jbrofuzz / XML Injection
- jbrofuzz / XPath Injection
- jbrofuzz / XSS
- jbrofuzz / Zero Fuzzers

OK Cancel

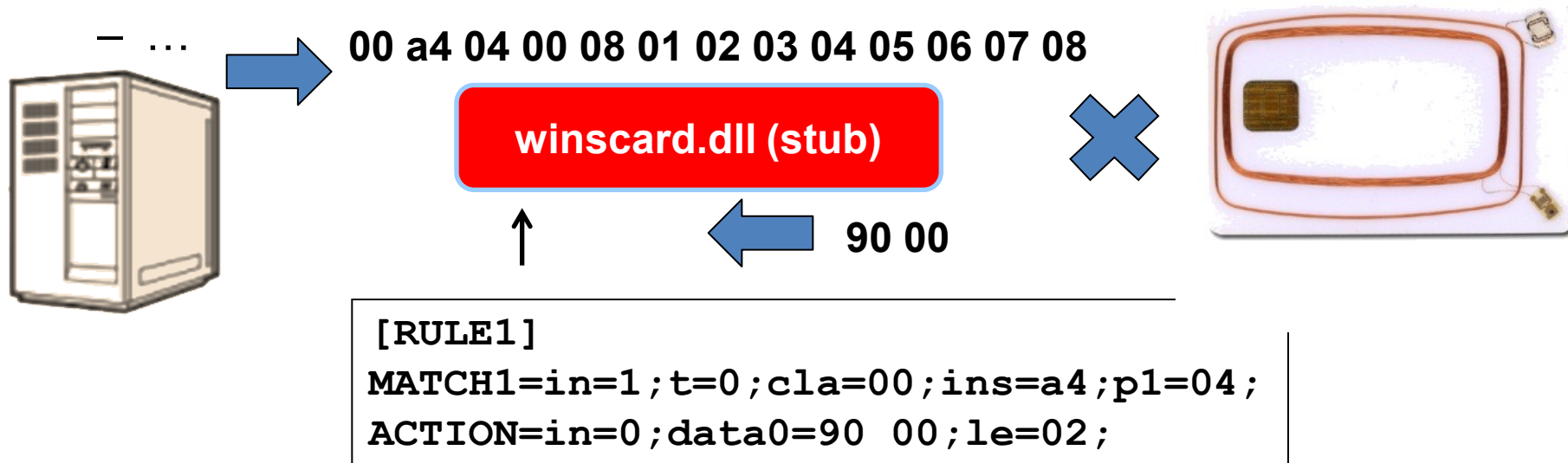
Forced Browse Fuzzer Params Http Sessions Zest Results Clients webSockets AJAX Spider Output



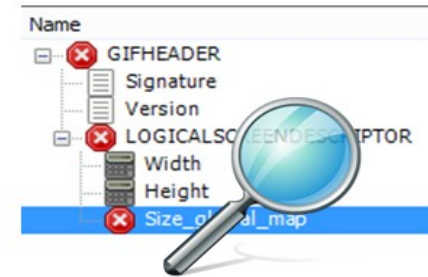


## APDUPlay - Smart card fuzzing

- Host to smart card communication done via PC/SC
- Custom winscard.dll stub written
- Manipulate incoming/outgoing APDUs
  - modify packet content
  - replay of previous packets



### 3. Validate data model



<http://iconarchive.com>,  
<http://awicons.com>,  
<http://www.pelfusion.com>

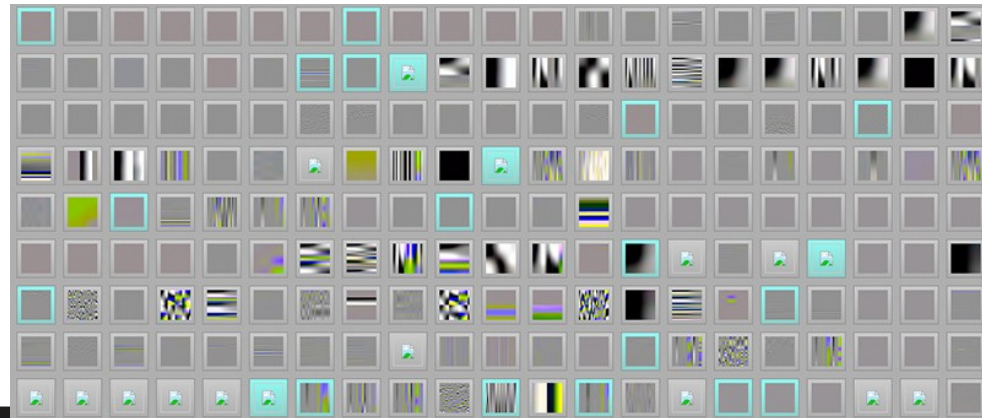
# VALIDATION





## American fuzzy lop

- Relatively new, but actively developed tool
- High speed fuzzer <http://lcamtuf.coredump.cx/afl/>
- Sophisticated generation of test cases (coverage)
- Automatic generation of input templates
  - E.g., valid JPEG image from “hello” string after few days
  - <http://lcamtuf.blogspot.cz/2014/11/pulling-jpegs-out-of-thin-air.html>
- Lots of real bugs found

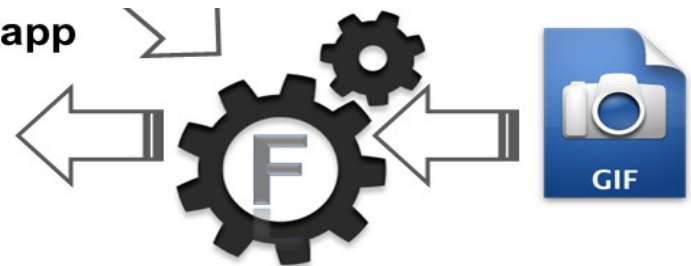


# Test coverage

- Random inputs have low coverage (usually)
  - Number of blocks visited in target binary
- Smart fuzzing tries to improve coverage
  - Way how to generate new inputs from existing
- E.g., Peach's minset tool
  - Gather a lot of inputs (files)
  - Run minset tool, traces with coverage stats are collected
  - Minimal set of files to achieve coverage is computed
  - Selected files are used as templates for fuzzing
- E.g. AFL fuzzer uses compile time instrumentation + genetic programming to create test cases



5. Send fuzzed input to app



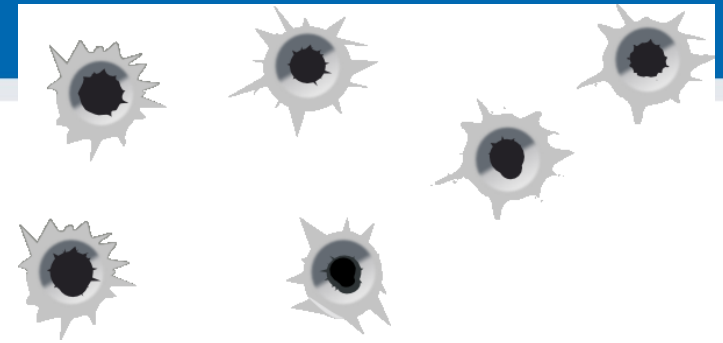
4. Generate fuzzed inputs



6. Monitor target app

<http://iconarchive.com>,  
<http://awicons.com>,  
<http://www.pelfusion.com>

# START, GENERATE, MONITOR



## How to detect “hit”?

- Application crash, uncaught exception...
  - Clear faults, easy to detect
- Error returned
  - Some errors are valid response
  - Some errors are valid response only in selected states
- Input accepted even when it shouldn't be
  - E.g., packet with incorrect checksum or modified field
- Some operation performed in incorrect state
  - E.g., door open without proper authentication
- Application behavior is impaired
  - E.g., response time significantly increases
- ...



# Peach monitors

## Windows Monitors

- [Windows Debugger Monitor](#)
- [Cleanup Registry Monitor](#)
- [Page Heap Monitor](#)
- [Popup Watcher Monitor](#)
- [Windows Service Monitor](#)

## OS X Monitors

- [OS X Crash Wrangler Monitor](#)
- [OS X Crash Reporter Monitor](#)

## Linux Monitors

- [Linux Crash Monitor](#)

## Cross Platform Monitors

- [CanaKit Relay Monitor](#)
- [Cleanup Folder Monitor](#)
- [IpPower9258 Monitor](#)
- [Memory Monitor](#)
- [Pcap Network Monitor](#)
- [Ping Monitor](#)
- [Process Launcher Monitor](#)
- [Process Killer Monitor](#)
- [Save File Monitor](#)
- [Socket Listener Monitor](#)
- [SSH Monitor](#)
- [SSH Downloader Monitor](#)
- [Vmware Control Monitor](#)





## 7. Analyze logs

*<http://iconarchive.com>,  
<http://awicons.com>,  
<http://www.pelfusion.com>*

# ANALYZE

# What to do with hit results?

- *Time intensive part of fuzzing*
- Not all hits are relevant (at least at the beginning)
  - Crashes by values not controllable by an attacker
  - !exploitable <https://msecdbg.codeplex.com/>
- Hits reproduction
  - Hit can be result of cumulative series of operations
- Many hits are duplicates
  - Inputs are different, but hit caused in the same part of code
- (Automatic) Bucketing of hits
  - E.g., Peach performs bucking based on signature of callstack

## Summary

- Fuzzers are cheap way to detect simpler bugs
  - If you don't use it, others will
- Try to find tool that fits your particular scenario
  - Check activity of development, support
- Fuzzing frameworks can ease variety of setups
  - But bit steeper learning curve
- If fuzzing will not find any bugs, check your model
- Try it!

# TAINT ANALYSIS

# Taint analysis

- Form of flow analysis
- Follow propagation of sensitive values inside program
  - e.g., user input that can be manipulated by an attacker
  - find all parts of program where value can “reach”
- *“Information **flows** from object  $x$  to object  $y$ , denoted  $x \rightarrow y$ , whenever information stored in  $x$  is transferred to, object  $y$ .” D. Denning*
- Native support in some languages (Ruby, Perl)
  - But not C++/Java ☹

## Taint sources

- Files (\*.pdf, \*.doc, \*.js, \*.mp3...)
  - User input (keyboard, mouse, touchscreen)
  - Network traffic
  - USB devices
  - ...
- 
- Every time there is information flow from value from untrusted source to other object X, object X is *tainted* – labeled as “tainted”

# Conclusions

- Dynamic analyzers can profile application
  - and find bugs not found by static analysis
- Fuzzing is “cheap” blackbox approach via malformed inputs

Questions 

# References

- Some books available, but...
- Michael Eddington, Demystifying fuzzers
  - Comparison of open-source tools, cost of adoption
  - BlackHat 2009, <https://www.blackhat.com/presentations/bh-usa-09/EDDINGTON/BHUSA09-Eddington-DemystFuzzers-PAPER.pdf>
  - <https://www.blackhat.com/presentations/bh-usa-09/EDDINGTON/BHUSA09-Eddington-DemystFuzzers-SLIDES.pdf>
  - RSA Conference 2010 talk  
<https://www.youtube.com/watch?v=Bm3Mfndrl1Y>
- OWASP fuzzing guidelines
  - <https://www.owasp.org/index.php/Fuzzing>
  - [https://www.owasp.org/index.php/OWASP\\_Testing\\_Guide\\_Appendix\\_C:\\_Fuzz\\_Vectors](https://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors)
- Tutorials and research papers on fuzzing <http://fuzzing.info/papers/>



# Peach tutorials

- Basic usage against vulnserver
  - <http://rockfishsec.blogspot.ch/2014/01/fuzzing-vulnserver-with-peach-3.html>
- Advanced tutorial (ZIP format fuzzing) – very good
  - <http://www.flinkd.org/2011/07/fuzzing-with-peach-part-1/>
- Tutorial for RAR fuzzing
  - <http://www.flinkd.org/2011/11/fuzzing-with-peach-part-2-fixups-2/>

## References

- MS post on Test coverage by fuzzing
  - <http://blogs.technet.com/b/srd/archive/2010/02/24/using-code-coverage-to-improve-fuzzing-results.aspx>
- Application and file fuzzing
  - <http://resources.infosecinstitute.com/application-and-file-fuzzing/>
- How I Learned to Stop Fuzzing and Find More Bugs
  - <https://www.defcon.org/images/defcon-15/dc15-presentations/dc-15-west.pdf>



# DYNAMIC ANALYSIS - PROFILING (WHITEBOX)

## Automatic measurement - profiling

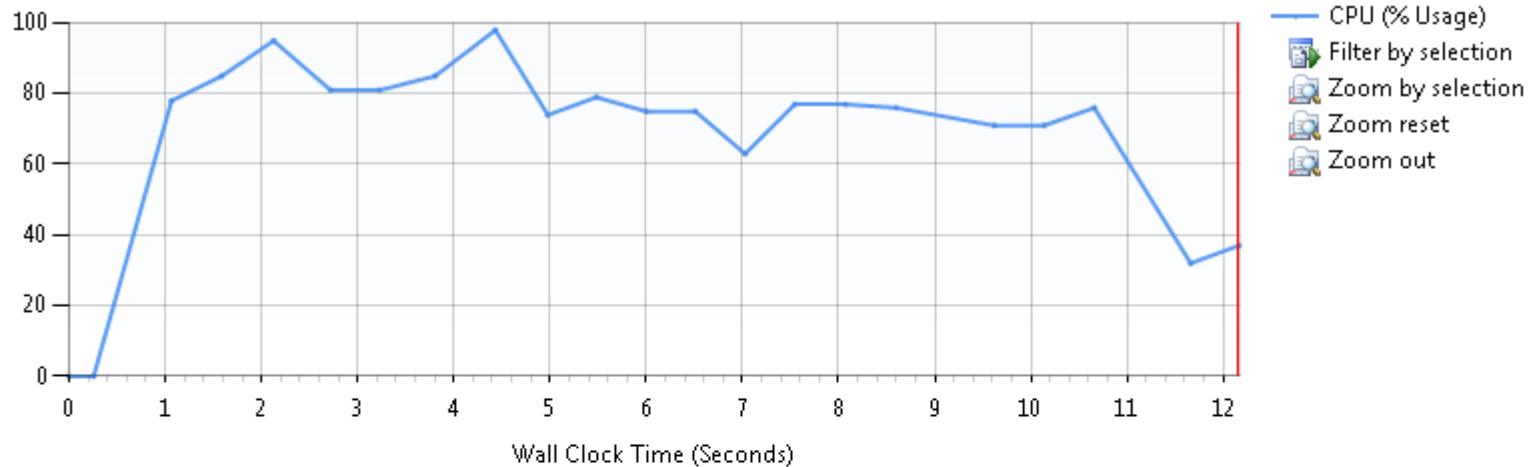
- Automatic tool to measure time and memory used
- “Time” spend in specific function
- How often a function is called
- Call tree
  - what function called actual one
  - based on real code execution (condition jumps)
- Many other statistics, depend on the tools
- Helps to focus and scope security analysis

# MS Visual Studio Profiler

- Analyze → Launch Performance Wizard
- Profiling method: **CPU Sampling**
  - check periodically what is executed on CPU
  - accurate, low overhead
- Profiling method: **Instrumentation**
  - automatically inserts special accounting code
  - will return exact function call counter
  - (may affect performance timings a bit)
    - additional code present
- May require admin privileges (will ask)

# MS VS Profiler – results (Summary)

- Where to start the optimization work?



## Hot Path

The most expensive call path based on sample counts

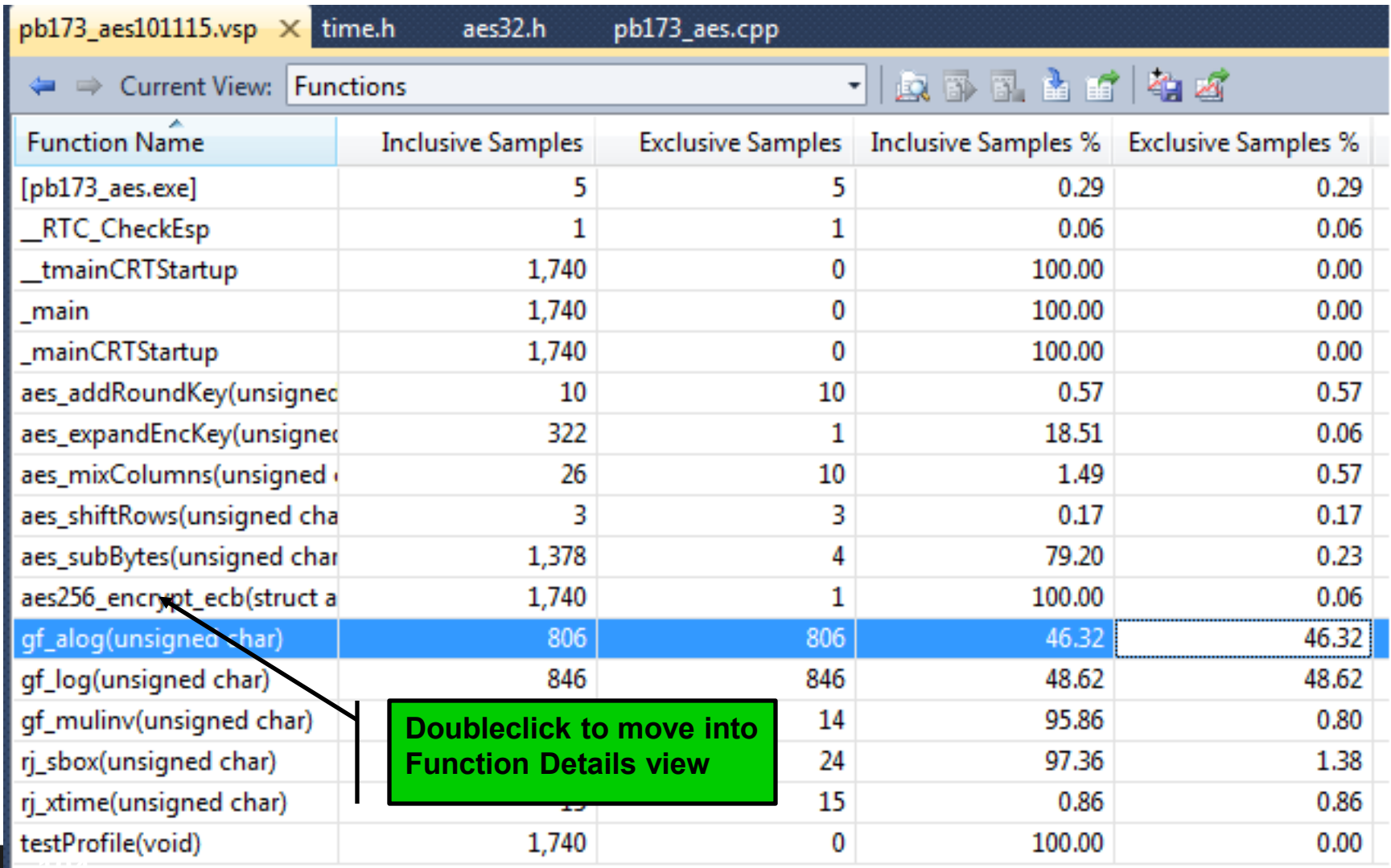
Name	Inclusive %	Exclusive %
↪ aes_subBytes(unsigned char *)	79.20	0.23
↪ rj_sbox(unsigned char)	78.97	1.26
↪ gf_mulinv(unsigned char)	77.59	0.75
🔥 gf_log(unsigned char)	39.43	39.43
🔥 gf_alog(unsigned char)	37.30	37.30

## MS VS Profiler – results (Functions)

- Result given in number of sampling hits
  - meaningful result is % of total time spend in function
- **Inclusive** sampling
  - samples hit in function or its children
  - aggregate over call stack for given function
- **Exclusive** sampling
  - samples hit in exclusively in given function
  - usually what you want
    - fraction of time spend in function code (not in subfunctions)



# MS VS Profiler – results (Functions)



pb173\_aes101115.vsp x time.h aes32.h pb173\_aes.cpp

Current View: Functions

Function Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
[pb173_aes.exe]	5	5	0.29	0.29
__RTC_CheckEsp	1	1	0.06	0.06
__tmainCRTStartup	1,740	0	100.00	0.00
_main	1,740	0	100.00	0.00
_mainCRTStartup	1,740	0	100.00	0.00
aes_addRoundKey(unsigned char)	10	10	0.57	0.57
aes_expandEncKey(unsigned char)	322	1	18.51	0.06
aes_mixColumns(unsigned char)	26	10	1.49	0.57
aes_shiftRows(unsigned char)	3	3	0.17	0.17
aes_subBytes(unsigned char)	1,378	4	79.20	0.23
aes256_encrypt_ecb(struct aes256_key)	1,740	1	100.00	0.06
gf_alog(unsigned char)	806	806	46.32	46.32
gf_log(unsigned char)	846	846	48.62	48.62
gf_mulinv(unsigned char)	14	14	95.86	0.80
rj_sbox(unsigned char)	24	24	97.36	1.38
rj_xtime(unsigned char)	15	15	0.86	0.86
testProfile(void)	1,740	0	100.00	0.00

Doubleclick to move into Function Details view

# GCC gcov tool

- <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov>
- 1. Compile program by GCC with additional flags
  - `gcc -Wall -fprofile-arcs -ftest-coverage main.c`
  - `gcc -Wall --coverage main.c`
  - additional monitoring code is added to binary
- 2. Execute program
  - files with “.bb” “.bbg” and “.da” extension are created
- 3. Analyze resulting files with **gcov**
  - `gcov main.c`
  - annotated source code is created
- Lcov - graphical front-end for gcov
  - <http://ltp.sourceforge.net/coverage/lcov.php>

# LCOV - code coverage report

Current view: [top level](#) - [example/methods](#) - [iterate.c](#) ([source](#) / [functions](#))

Test: [Basic example](#) ( [view descriptions](#) )

Date: 2012-10-12

Legend: Lines: hit not hit | Branches: + taken - not taken # not executed

	Hit	Total	Coverage
Lines:	8	8	100.0 %
Functions:	1	1	100.0 %
Branches:	4	4	100.0 %

Branch data	Line data	Source code
1	:	: /*
2	:	: * methods/iterate.c
3	:	: *
4	:	: * Calculate the sum of a given range of integer numbers.
5	:	: *
6	:	: * This particular method of implementation works by way of brute force,
7	:	: * i.e. it iterates over the entire range while adding the numbers to finally
8	:	: * get the total sum. As a positive side effect, we're able to easily detect
9	:	: * overflows, i.e. situations in which the sum would exceed the capacity
10	:	: * of an integer variable.
11	:	: *
12	:	: */
13	:	:
14	:	: #include <stdio.h>
15	:	: #include <stdlib.h>
16	:	: #include "iterate.h"
17	:	:
18	:	:
19	:	3 : int iterate_get_sum (int min, int max)
20	:	: {
21	:	:     int i, total;
22	:	:
23	:	3 :     total = 0;
24	:	:
25	:	:     /* This is where we loop over each number in the range, including
26	:	:     both the minimum and the maximum number. */
27	:	:
28	[ + + ]:	67548 :     for (i = min; i <= max; i++)
29	:	:     {
30	:	:         /* We can detect an overflow by checking whether the new
31	:	:         sum would become negative. */
32	:	:
33	[ + + ]:	67546 :     if (total + i < total)
34	:	:     {
35	:	1 :         printf ("Error: sum too large!\n");
36	:	1 :         exit (1);
37	:	:     }
38	:	:
39	:	:     /* Everything seems to fit into an int, so continue adding. */
40	:	:
41	:	67545 :     total += i;

Taken from: <http://ltp.sourceforge.net/coverage/lcov/output/example/methods/iterate.c.gcov.html>

# REVERSE ENGINEERING (BLACKBOX)

# Reverse engineering

- Art of discovering principles through analysis of structure, functions and operation
- Legality
  - Own binary without documentation
  - Interoperability
  - Anti-virus research
  - Fair use, education
  - Forensics
- Problem with recent copyright laws
  - even attempt to circumvent is illegal
  - not only selling circumvented content

## Disassembler vs. debugger

- Static vs. dynamic code analysis
- Debugger vs. Debugger with advanced modification tools (Visual Studio vs. OllyDbg)
- Assembler vs. bytecode
  - Instruction set
  - Register-based vs. stack-based execution

## Lena tutorials

- Nice introduction tutorials for reversing/cracking
- Win32 binary
  - Lena tutorials 1 and 2
- Name of the registers
  - (EAX 32bit, AX 16bit, AH/AL 8bit)
- Registers (FPU):
  - Z – zero flag, C – carry flag, S – sign flag
  - EIP ... next address to execute (instruction pointer)
  - EBX ... usually loop counter

## Startup resources

- The Reverse Code Engineering Community:  
<http://www.reverse-engineering.net/>
- Tutorials for You: <http://www.tuts4you.com>
- RE on Wikipedia:  
[http://en.wikipedia.org/wiki/Reverse\\_engineering](http://en.wikipedia.org/wiki/Reverse_engineering)



## Disassembling binary code

- Interactive Disassembler is legendary full-fledged disassembler with ability to disassemble many different platforms.
  - Free version available for non-commercial uses
  - Free version disassemble only Windows binaries
  - <http://www.hex-rays.com/idapro/idadownfreeware.htm>
- Very nice visualization and debugger feature (similar as OllyDbg)
  - Try it!

# Decompiling binary code

- Decompiler is able to produce source code from binary code. Decompiler needs to do disassembling first and then try to create code that will in turn produce binary code you have at the beginning.
- Resulting code will NOT contain information removed during compilation
  - (comments, function names, formatting...)
  - Read <http://www.debugmode.com/dcompile/> for more info
- Still can be of great help
- Problem to find well working free disassembler
  - [http://en.wikibooks.org/wiki/X86\\_Disassembly/Disassemblers\\_and\\_Decompile](http://en.wikibooks.org/wiki/X86_Disassembly/Disassemblers_and_Decompile)

## Resources

- The Reverse Code Engineering Community:  
<http://www.reverse-engineering.net/>
- Tutorials for You: <http://www.tuts4you.com>
- Disassembling tutorial  
<http://www.codeproject.com/KB/cpp/reversedisasm.aspx>

# OLD FUZZING

## Example with many possible inputs

- Multiple inputs to application
- Not possible to evaluate manually
  - or done very frequently: UT, TDD, continuous integration
- Sometimes not possible to bruteforce at all
  - to many combinations
  - usual situation!
- Easy to overlook potential problem

# Fuzzer - principle

1. Send high number of different inputs into application
    - user interface (UI), command line options, import/export capabilities
  2. Inputs generated randomly or according to predefined pattern
    - Protocol/file-format/data-type dependant
    - E.g., input string with different lengths (1MB user name)
    - E.g., valid input with certain percentage of random modifications (jpg file with random changes)
  3. Fuzzer monitors application for crash or emitted error
    - Memory corruption, invalid state...
    - When detected, problem is manually inspected
- <https://www.owasp.org/index.php/Fuzzing>

# Fuzzing – advantages/disadvantages

- Fuzzers tend to find simple bugs
  - more protocol-aware fuzzer is, less weird problems will find
- Fuzzing advantages
  - very simple design
  - allow to find bugs missed by human eye
  - sometimes only way to test (completely closed system)
- Fuzzing disadvantages
  - increased difficulty to evaluate impact/dangerosity
    - closed system is evaluated, black box testing

# Types of fuzzing

- Application fuzzing
  - generates inputs for application (stdin, memory...)
- Protocol fuzzing
  - manipulation of protocol level
- File format fuzzing
  - generates malformed file samples
  - if program crashes, debug log is created
  - attack against parser layer
  - attack against codec/application layer
  - example: [MS04-028](#) Microsoft's JPEG GDI+ vulnerability
    - <http://technet.microsoft.com/en-us/security/bulletin/ms04-028>



## Fuzzing – approaches

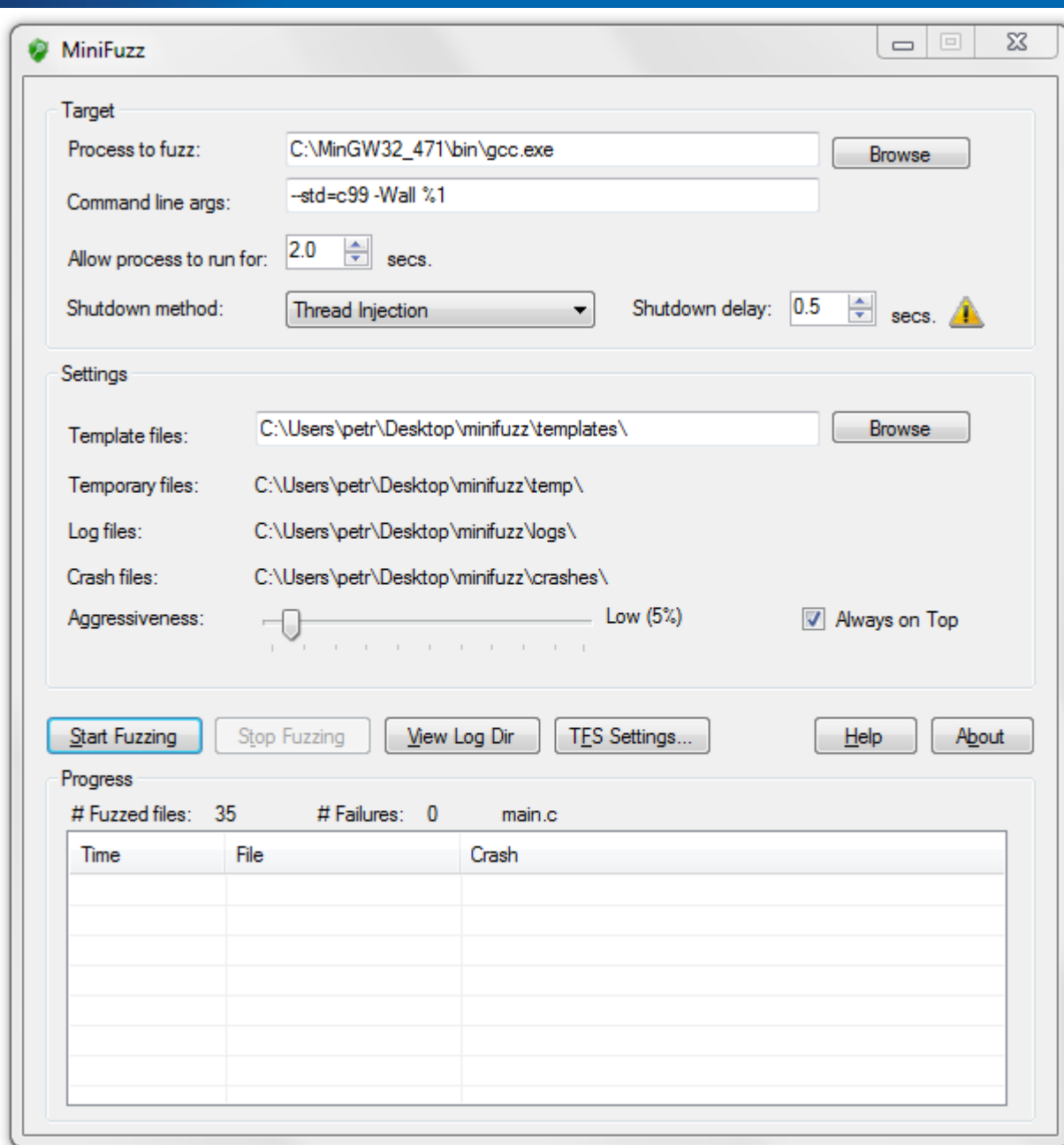
- Capture valid inputs and modify some bytes
  - randomly
  - according to given regular expression
  - Random changes with post-processing (e.g., correct CRC)
- Binary vs. text oriented fuzzing

# Available tools

- Microsoft's SDL MiniFuzz File Fuzzer
- Microsoft's SDL Regex Fuzzer
- Ilja van Sprundel's mangle.c
  - [https://ext4.wiki.kernel.org/index.php/Filesystem\\_Testing\\_Tools/mangle.c](https://ext4.wiki.kernel.org/index.php/Filesystem_Testing_Tools/mangle.c)
  - filename and header size
  - change between 0 and 10% of header with random bytes
  - example data
- zzuf - multi-purpose fuzzer
  - application input fuzzer
  - intercepting file and network operations and changing random bits in the program's input
  - <http://sam.zoy.org/zzuf/>

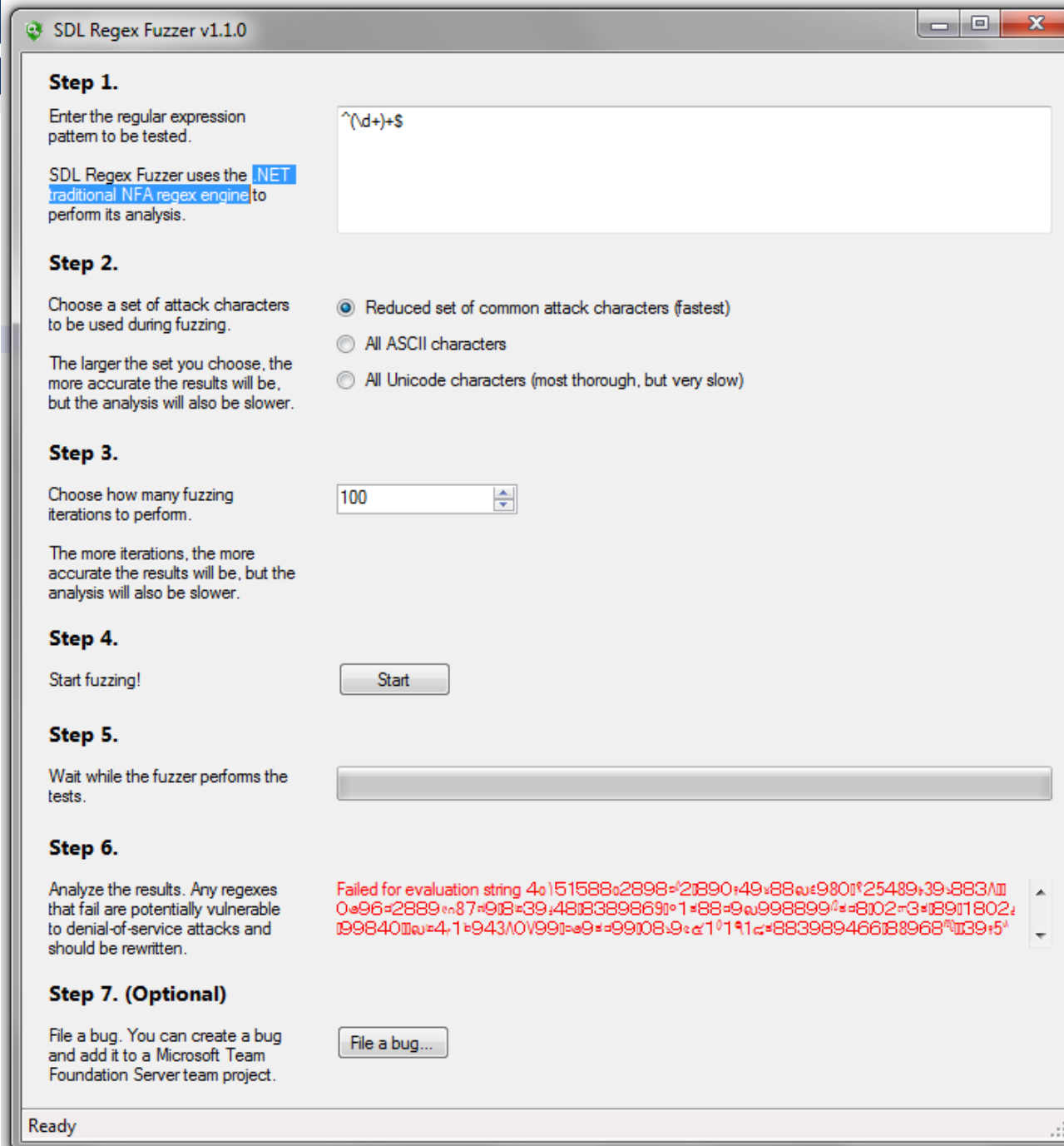
# Microsoft's SDL MiniFuzz File Fuzzer

- Application input files fuzzer
  - <http://www.microsoft.com/en-us/download/details.aspx?id=21769>
  - UsingMiniFuzz.htm
- Templates for valid input files (multiple)
- Modify valid input file (randomly, % aggressiveness)
- Run application with partially modified inputs
- Log resulting crash (if happen)
  - exception, CPU registers...
- Can be incorporated directly into Visual Studio
- Video overview
  - <http://msdn.microsoft.com/en-us/security/gg675011.aspx>



## Microsoft's SDL Regex Fuz

- Test of regular expressions evaluations
- May cause denial-of-service attack
- Use when your program use regex evaluation
- Video overview
  - <http://msdn.microsoft.com/en-us/security/gg675012.aspx>
- <http://blogs.msdn.com/b/sdl/archive/2010/10/12/new-tool-sdl-regex-fuzzer.aspx>
- Example: `^\d+$`



# SPIKE

- Tool for fuzzing analysis of network protocols
  - <http://www.immunitysec.com/resources-freesoftware.shtml>
- Overview of SPIKE capabilities
  - <https://www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt>
- Fuzzing tutorial with SPIKE on Vulnserver
  - <http://resources.infosecinstitute.com/intro-to-fuzzing/>
  - Windows & Linux version
- Another SPIKE tutorial
  - <http://pentest.cryptocity.net/fuzzing/>