PA193 - Secure coding principles and practices

Static analysis with annotations, dynamic analysis, security testing

Petr Švenda svenda@fi.muni.cz

CROCS

Centre for Research on
Cryptography and Security

Overview

- Lecture:
  - Annotations for static analysis (Microsoft SAL)
  - blackbox / whitebox testing
  - whitebox testing – how to analyze large code base
  - fuzzing (blackbox testing)
  - tools
- Labs

CODE ANNOTATIONS – MS SAL

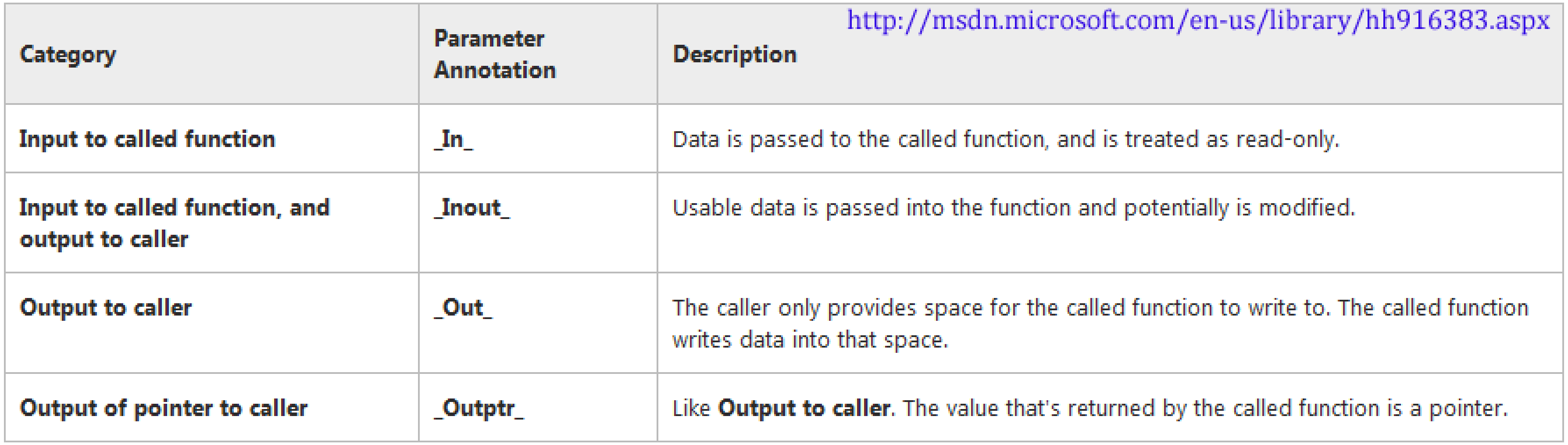
Microsoft SDL C/C++ static checker

- http://msdn.microsoft.com/en-us/security/gg675036
- Problems not found by PREfast checker by default
- But solvable by SAL
  - check of return value
  - argument must be not NULL
  - string must be terminated
  - length of data read / written into buffer

Source-code annotation language (SAL)

- Microsoft's annotation language
- Improves code checking done by MS static checker PREfast
- MS Annotations
  - http://msdn.microsoft.com/en-us/library/ms182032.aspx
  - http://msdn.microsoft.com/en-us/library/hh916382.aspx
- Additional requirements are added to declaration of function, structure... via (non-standard) keywords
- Validity of such requirements are checked by PREfast
  - in pre-state (before fnc call) & in post-state (after fnc call)

SAL functions basics



http://msdn.microsoft.com/en-us/library/hh916383.aspx

Category	Parameter Annotation	Description
Input to called function	**_In_**	Data is passed to the called function, and is treated as read-only.
Input to called function, and output to caller	**_Inout_**	Usable data is passed into the function and potentially is modified.
Output to caller	**_Out_**	The caller only provides space for the called function to write to. The called function writes data into that space.
Output of pointer to caller	**_Outptr_**	Like **Output to caller**. The value that's returned by the called function is a pointer.

- Optional version of arguments
  - argument might be NULL
  - `_In_opt, _Out_opt...`
  - function must perform check before use

SAL functions basics II.

- Pointer type annotations
- `_Outptr_`
  - should not be NULL
  - should be initialized
- _Outptr_opt_
  - can be NULL, must be checked

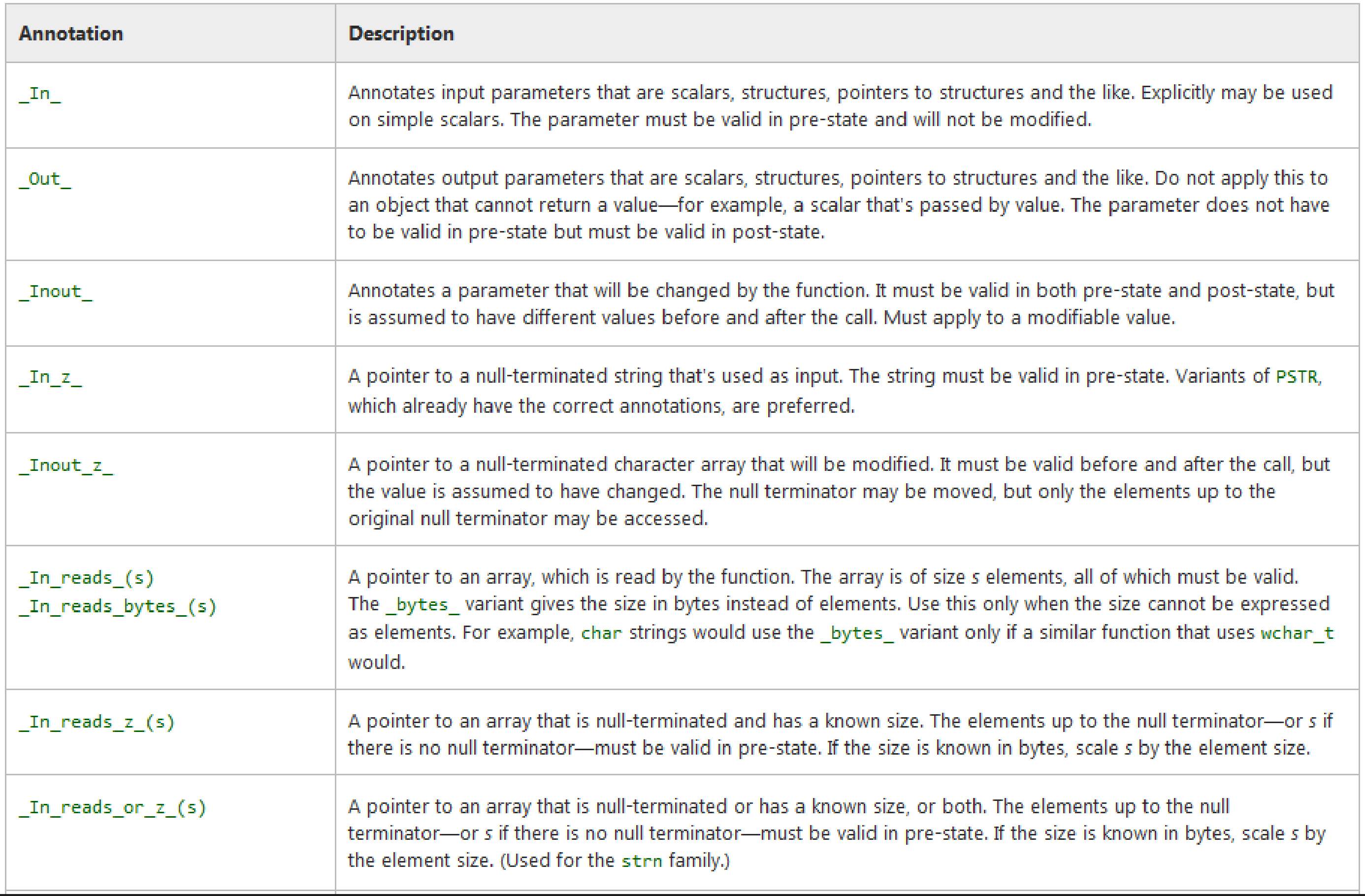
```
void salDemo(_In_ int* pInArray, _Outptr_ int** ppArray) {
}

int main(int argc, char* argv[]) {
        int* pArray = NULL;
        int* pArray2 = NULL;
        if (strcmp(argv[1], "alloc") == 0) pArray = new int[5];
        salDemo(pArray, &pArray2);

        return 0;
}
```

- PREfast analysis

```
test.cpp(34): warning : C6101: Returning uninitialized memory '*ppArray'. A successful
   path through the function does not set the named _Out_ parameter.
test.cpp(49): warning : C6387: 'pArray' could be '0':  this does not adhere to the
   specification for the function 'salDemo'.
test.cpp(49): warning : C6001: Using uninitialized memory '*pArray'.
```

SAL annotations - functions

- http://msdn.microsoft.com/en-us/library/hh916382.aspx



Annotation	Description
`_In_`	Annotates input parameters that are scalars, structures, pointers to structures and the like. Explicitly may be used on simple scalars. The parameter must be valid in pre-state and will not be modified.
`_Out_`	Annotates output parameters that are scalars, structures, pointers to structures and the like. Do not apply this to an object that cannot return a value—for example, a scalar that's passed by value. The parameter does not have to be valid in pre-state but must be valid in post-state.
`_Inout_`	Annotates a parameter that will be changed by the function. It must be valid in both pre-state and post-state, but is assumed to have different values before and after the call. Must apply to a modifiable value.
`_In_z_`	A pointer to a null-terminated string that's used as input. The string must be valid in pre-state. Variants of `PSTR`, which already have the correct annotations, are preferred.
`_Inout_z_`	A pointer to a null-terminated character array that will be modified. It must be valid before and after the call, but the value is assumed to have changed. The null terminator may be moved, but only the elements up to the original null terminator may be accessed.

`_In_reads_(s)`
`_In_reads_bytes_(s)`
	A pointer to an array, which is read by the function. The array is of size *s* elements, all of which must be valid. The `_bytes_` variant gives the size in bytes instead of elements. Use this only when the size cannot be expressed as elements. For example, `char` strings would use the `_bytes_` variant only if a similar function that uses `wchar_t` would.

`_In_reads_z_(s)`	A pointer to an array that is null-terminated and has a known size. The elements up to the null terminator—or *s* if there is no null terminator—must be valid in pre-state. If the size is known in bytes, scale *s* by the element size.
`_In_reads_or_z_(s)`	A pointer to an array that is null-terminated or has a known size, or both. The elements up to the null terminator—or *s* if there is no null terminator—must be valid in pre-state. If the size is known in bytes, scale *s* by the element size. (Used for the `strn` family.)

SAL annotations –

- Structs and classes can be also annotated
  - http://msdn.microsoft.com/en-us/library/jj159528.aspx
- Locking behavior for concurrency can be annotated
  - http://msdn.microsoft.com/en-us/library/hh916381.aspx
- Whole function can be annotated
  - http://msdn.microsoft.com/en-us/library/jj159529.aspx
  - **`_Must_inspect_result_`**
- Best practices
  - http://msdn.microsoft.com/en-us/library/jj159525.aspx

SAL – examples (in and out buffer)

```
// read from buffer, but outside
int readData( void *buffer, int length );
int readData(_In_reads_(length) void *buffer, int length );

// writes specified amount of data into buffer
int fillData( void *buffer, const int maxLength, int *length );
int fillData(_Out_writes_all_(length) void *buffer, const int length );

// writes into buffer maxLength at max, but possibly less and modifies also length argument
int fillData( void *buffer, const int maxLength, int *length );
// Check if no more then maxLength and *length is written, also check range of length
int fillData( __Out_writes_to_( maxLength, *length ) void *buffer,
        const int maxLength, _Out_range_(0, maxLength-1) int *length );

// read AND write from buffer
int readWriteData( void *buffer, int length );
int readWriteData(_Inout_updates_(length) void *buffer, int length );
```

SAL – examples (pointers, strings)

```
// pass argument by value foo pointer
int getInfo( struct thing *thingPtr );
// value is used as input and output => _Inout_
int getInfo( _Inout_ struct thing *thingPtr );

// pass C null-terminated strings
int writeString( const char *string );
// must be null terminated string › _In_z_
int writeString( _In_z_ const char *string );
```

Using SAL with / without PREfast

- Cross-platform code
  - Compiled with MSVC for Windows (SAL is supported)
  - Compiled with GCC for Linux (SAL not supported)
- SAL annotations makes GCC compilation to fail
- Solution
  - Create custom `#define` for most common SAL annotations
  - Define as empty if not compiled with MSVC
  - Can be also tuned when SAL annotation changes itself
- Peter Gutmann' Experiences with SAL/PREfast
  - http://www.cs.auckland.ac.nz/~pgut001/pubs/sal.html

Wrapping defines for SAL

```
#if defined( _MSC_VER ) && defined( _PREFAST_ )
  #define IN_BUFSIZE           _In_reads
  #define IN_BUFSIZE_OPT       _In_reads_opt
  #define OUT_BUFSIZE          _Out_writes
  #define OUT_BUFSIZE_OPT      _Out_writes_opt
  #define OUT_PTR              _Outptr_
  #define OUT_PTR_OPT          _Outptr_opt_
#else
  #define IN_BUFSIZE( size )
  #define IN_BUFSIZE_OPT( size )
  #define OUT_BUFSIZE( max, size )
  #define OUT_BUFSIZE_OPT( max, size )
  #define OUT_PTR
  #define OUT_PTR_OPT
#endif /* VC++ with source analysis enabled */
```

Modification of http://www.cs.auckland.ac.nz/~pgut001/pubs/sal.html

Deputy (annotations for GCC)

- Not active any more ☹, last update 2006?
  - http://www.stanford.edu/class/cs295/asgns/asgn5/www/

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

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
- List of tools for dynamic analysis
  - https://en.wikipedia.org/wiki/Dynamic_program_analysis
- Most performance analyzers are dynamic analyzers
  - MS Visual Studio→Analyze→Start performance analysis
  - gcc -Wall -fprofile-arcs -ftest-coverage main.c

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 (later presentation)
- 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
- Writes into already de-allocated memory
  - Valgrind tries to defer reallocation of freed memory as long as possible to detect subsequent reads/writes here

```cpp
#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;
}
```

```cpp
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 /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 lea
  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

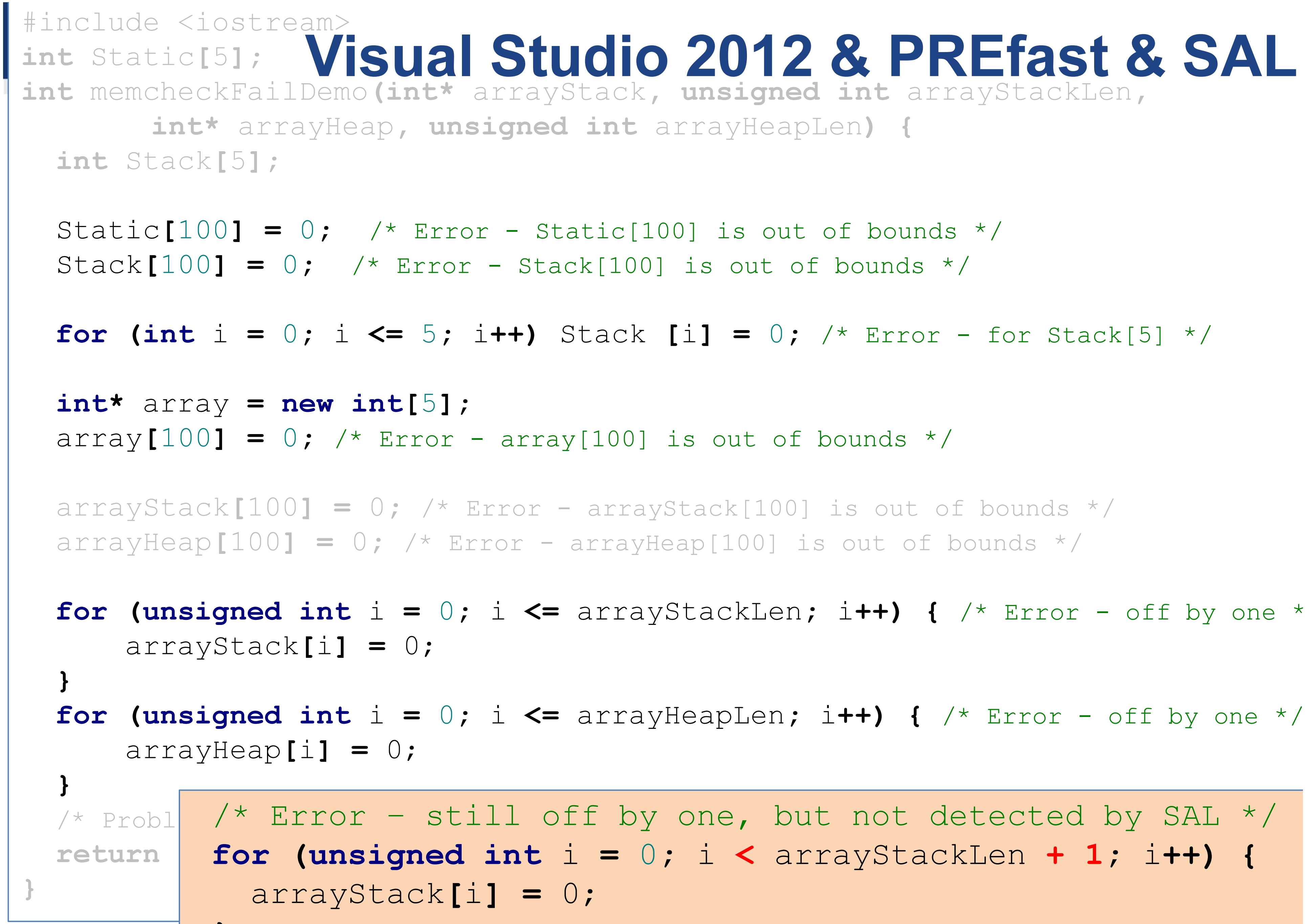
```
#include <iostream>
int Static[5];
int memcheckFailDemo(int* arrayStack, unsigned int arrayStackLen,
       int* arrayHeap, unsigned int arrayHeapLen) {
  int Stack[5];

  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
}
```

```
/* Error - still off by one, but not detected by SAL */
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:17)
==    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:363)
==    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-sgcheck

```
==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 0x7fefffe34 expected vs actual:
==15979==  Expected: stack array "Stack" of size 20 in this frame
==15979==  Actual:   unknown
==15979==  Actual:   is 0 after Expected
==15979==
==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 0x7fefffe74 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
{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
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

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→y , whenever information stored in x is transferred to, object y.” D. Denning*
- Native support in some languages (Ruby, Perl)
  - But not C++ ☹

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”

Execution of sensitive operation

- Before sensitive operation (e.g., system()) is executed with value, taint label is checked
  - if value is tainted, alert is issued
- Untrusted data reaching privilege location is detected
  - can detect even unknown attacks
  - (but sometimes we need to use user input)

Taint analysis - tools

- Taintgrind
  - http://www.cl.cam.ac.uk/~wmk26/taintgrind/
  - additional module to Valgrind
  - dynamic taint analyzer for C/C++
  - output memory traces (information flows) already produced by Valgrind
- Tanalysis
  - http://code.google.com/p/tanalysis/
  - static taint analyzer for C
  - plugin for Frama-C http://frama-c.com/
- Read more about taint analysis
  - http://users.ece.cmu.edu/~ejschwar/papers/oakland10.pdf

Microsoft PREfast + Taint analysis

- Warning C6029 is issued when tainted value is passed to parameter marked as [Post(Tainted=No)]
  - without any checking (any condition statement)
- http://msdn.microsoft.com/en-us/library/ms182047%28v=vs.100%29.aspx

```
// C
#include <CodeAnalysis\SourceAnnotations.h>
void f([SA_Pre(Tainted=SA_Yes)] int c);

// C++
#include <CodeAnalysis\SourceAnnotations.h>
using namespace vc_attributes;
void f([Pre(Tainted=Yes)] int c);
```

DEBUGGING SYMBOLS (WHITEBOX)

Release vs. Debug

- Optimizations applied (compiler-specific settings)
  - gcc –Ox (http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html)
    - -O0 no optimization (Debug)
    - -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

Debug symbols

- 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

DYNAMIC ANALYSIS - PROFILING (WHITEBOX)

Automatic measurement - profiling

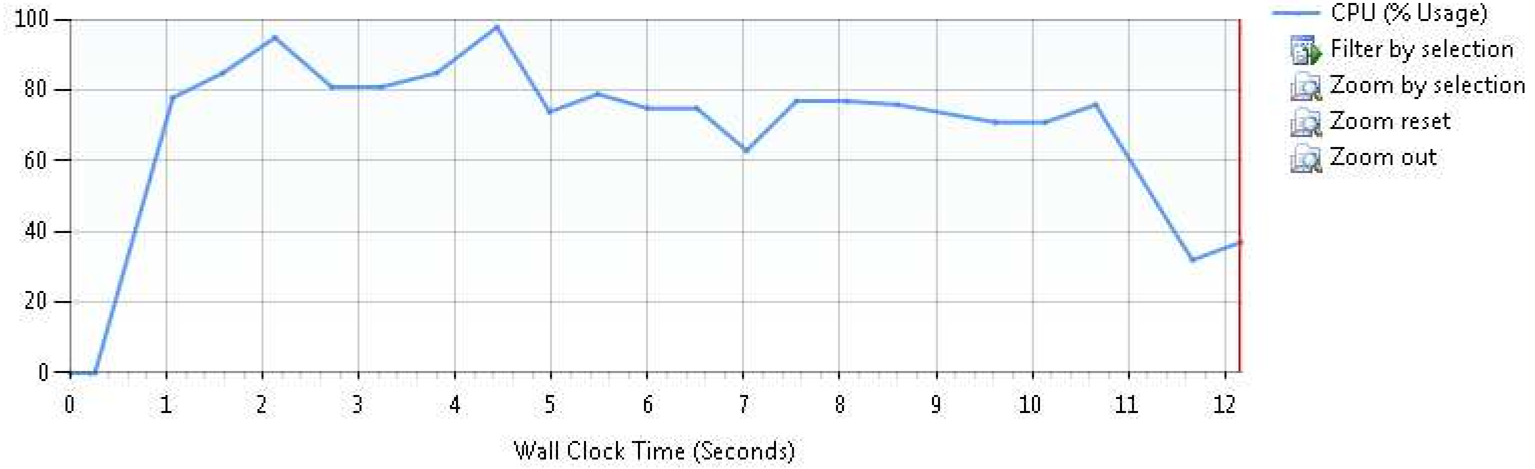
- Automatic tool to measure time and memory used
- “Time” spend in specific function
- How often a function is called
- Call tree
  - what function called actual one
  - based on real code execution (condition jumps)
- Many other statistics, depend on the tools
- Helps to focus and scope security analysis

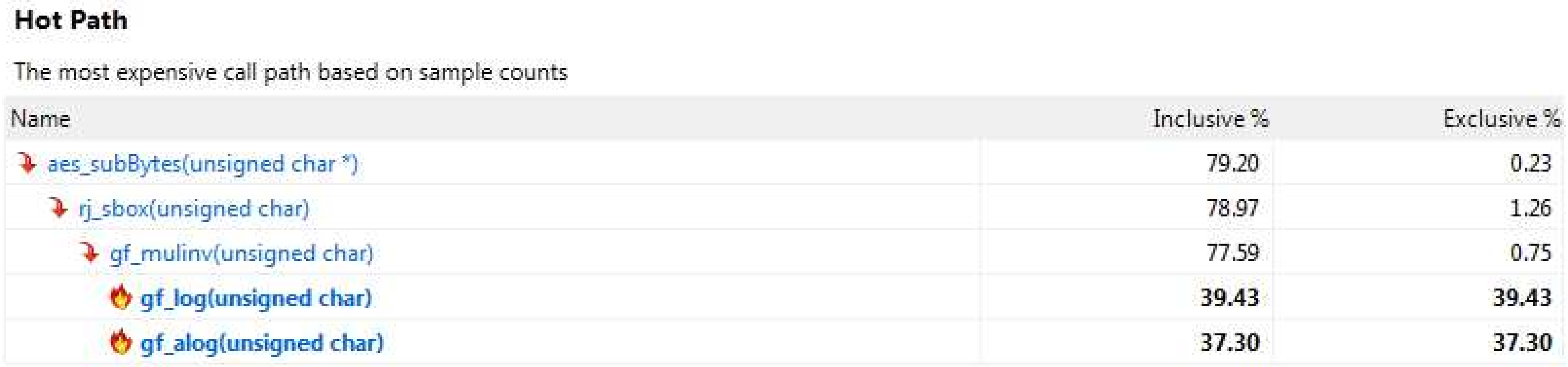
MS Visual Studio Profiler

- 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

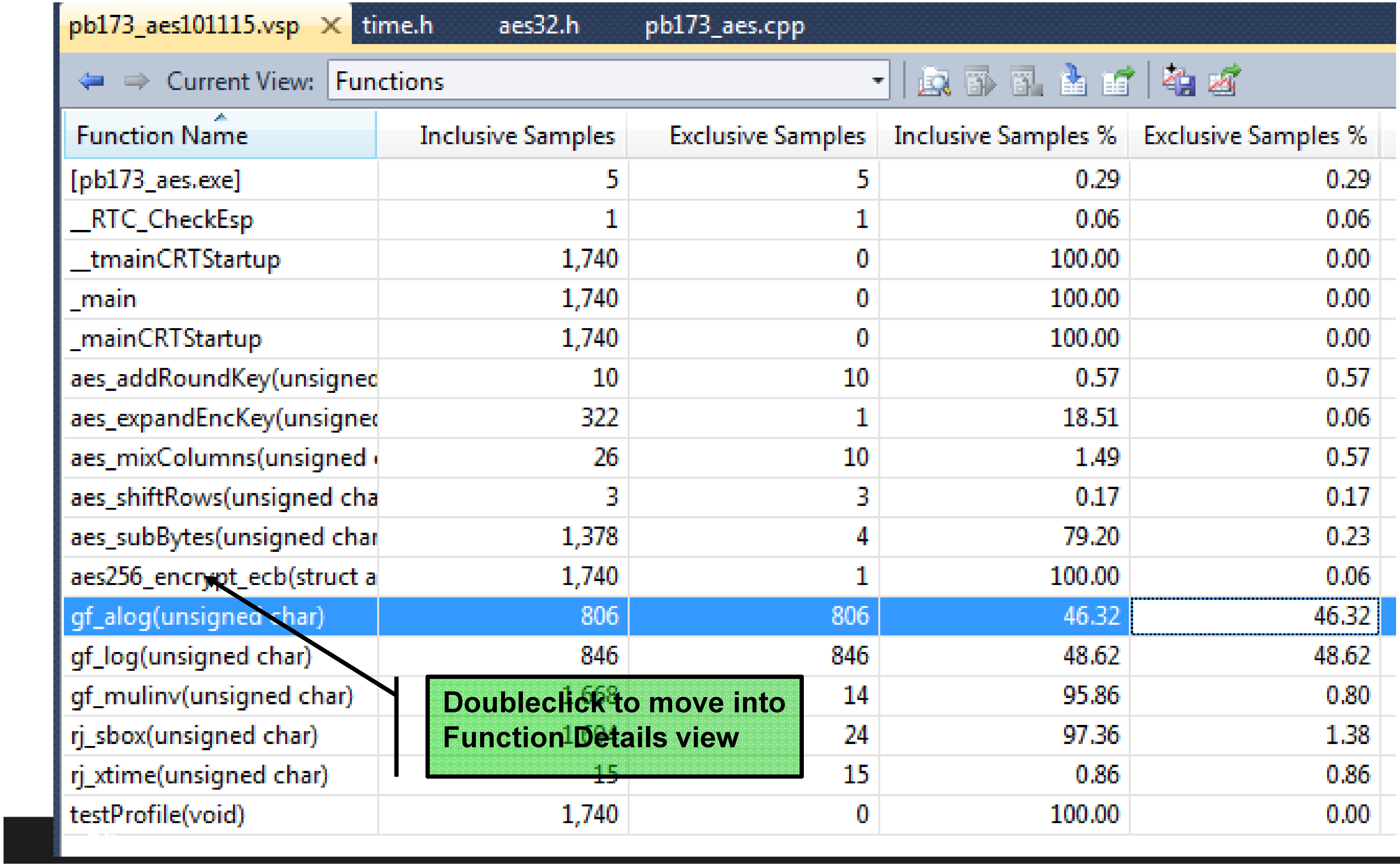
The most expensive call path based on sample counts

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)



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(unsignec	10	10	0.57	0.57
aes_expandEncKey(unsignec	322	1	18.51	0.06
aes_mixColumns(unsigned	26	10	1.49	0.57
aes_shiftRows(unsigned cha	3	3	0.17	0.17
aes_subBytes(unsigned char	1,378	4	79.20	0.23
aes256_encrypt_ecb(struct a	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)	1,668	14	95.86	0.80
rj_sbox(unsigned char)	1,694	24	97.36	1.38
rj_xtime(unsigned char)	15	15	0.86	0.86
testProfile(void)	1,740	0	100.00	0.00

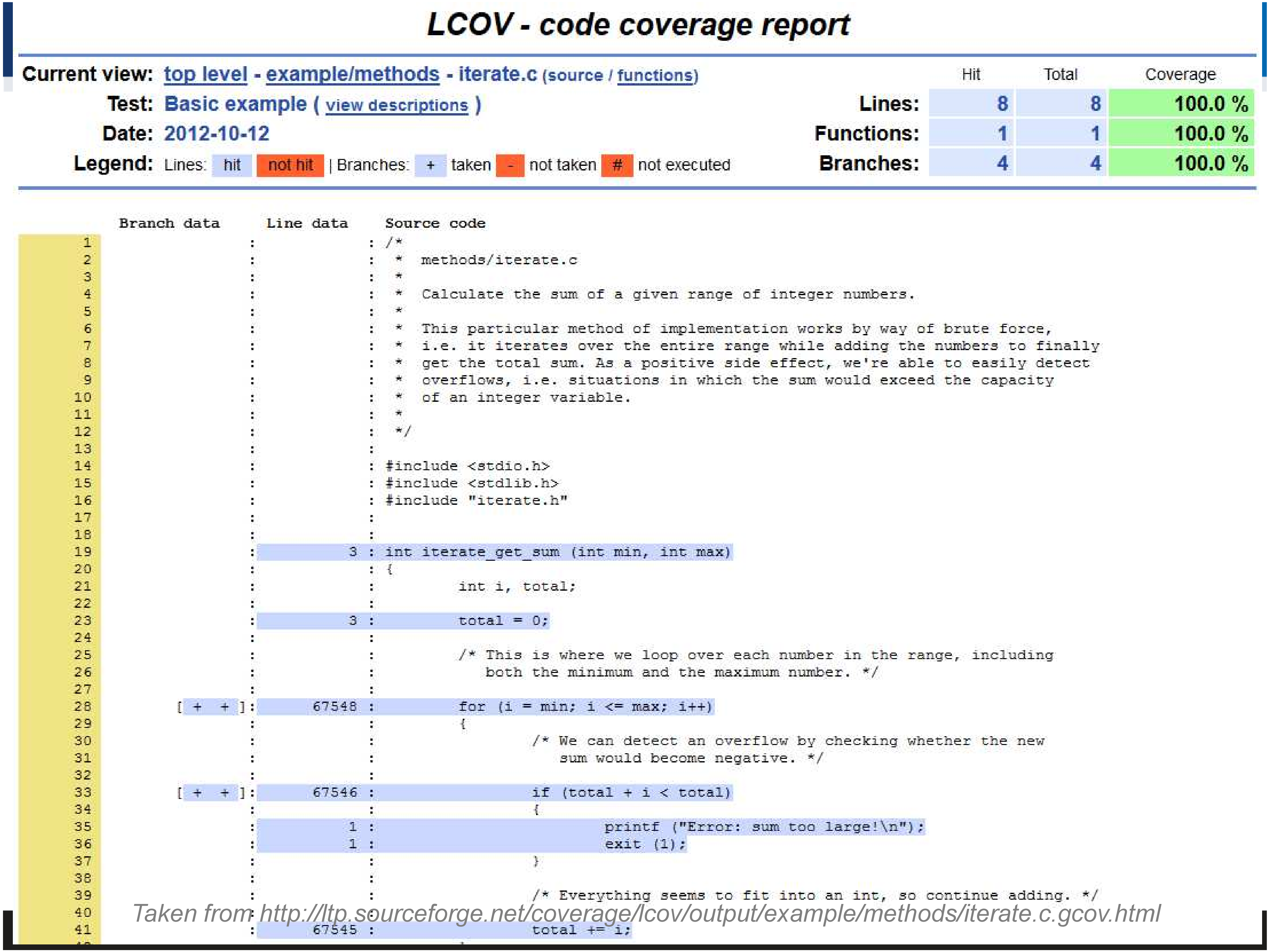
Doubleclick to move into Function Details view

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

FUZZING (BLACKBOX)

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

Fuzzing

- Attack vectors within application's I/O
  - user interface (UI)
  - command line options
  - import/export capabilities
- Fuzzers tend to find simple bugs
  - more protocol-aware fuzzer is, less weird problems will find
- Protocol/file-format dependant
- Data-type dependant
- https://www.owasp.org/index.php/Fuzzing

Fuzzing – advantages/disadvantages

- 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

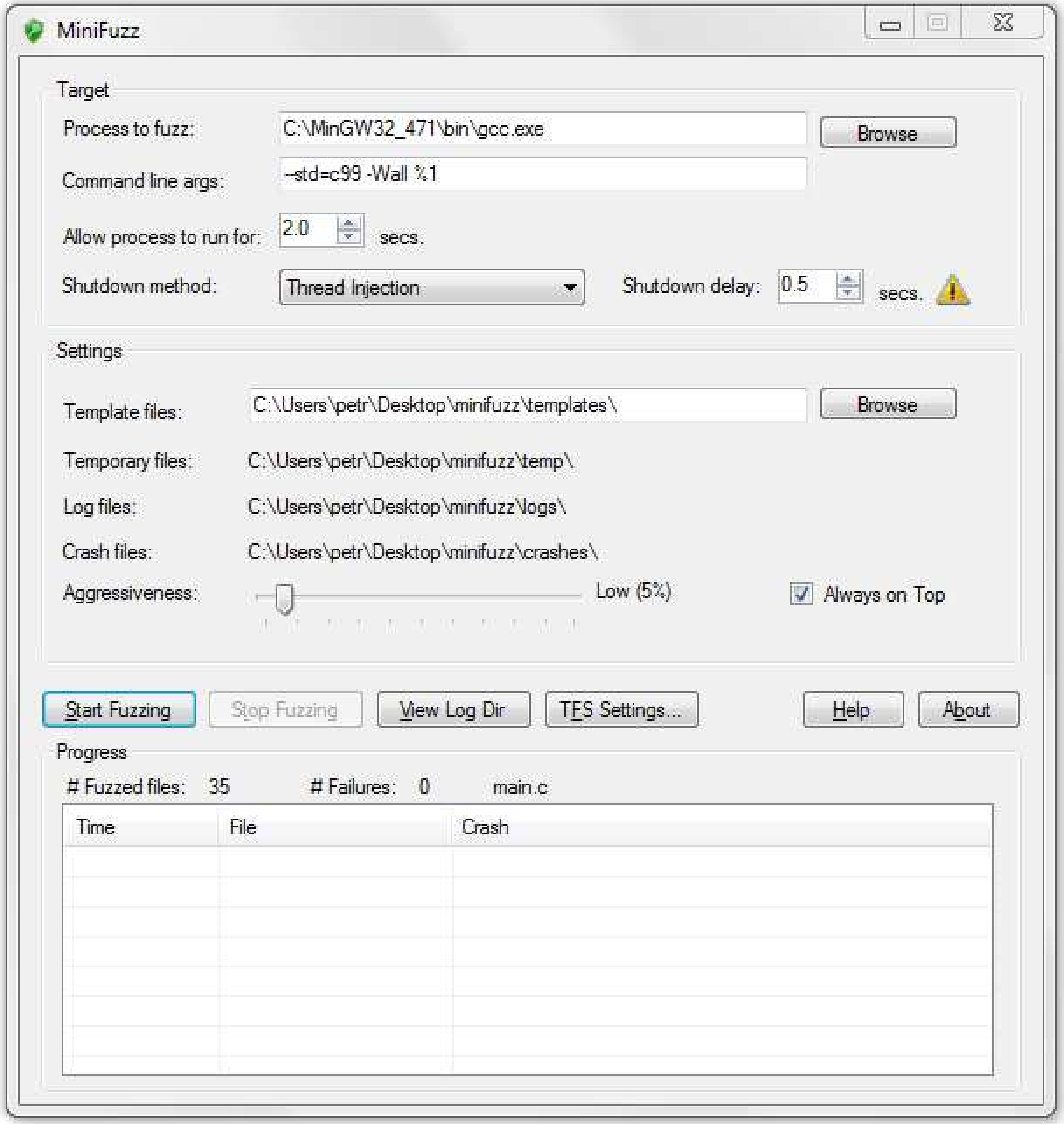
- Capture valid inputs and modify some bytes
  - randomly
  - according to given regular expression
- Binary vs. text oriented fuzzing

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
  - 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



MiniFuzz

Target

Process to fuzz: C:\MinGW32_471\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\petr\Desktop\minifuzz\templates\

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

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

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

Aggressiveness: Low (5%)

☑ Always on Top

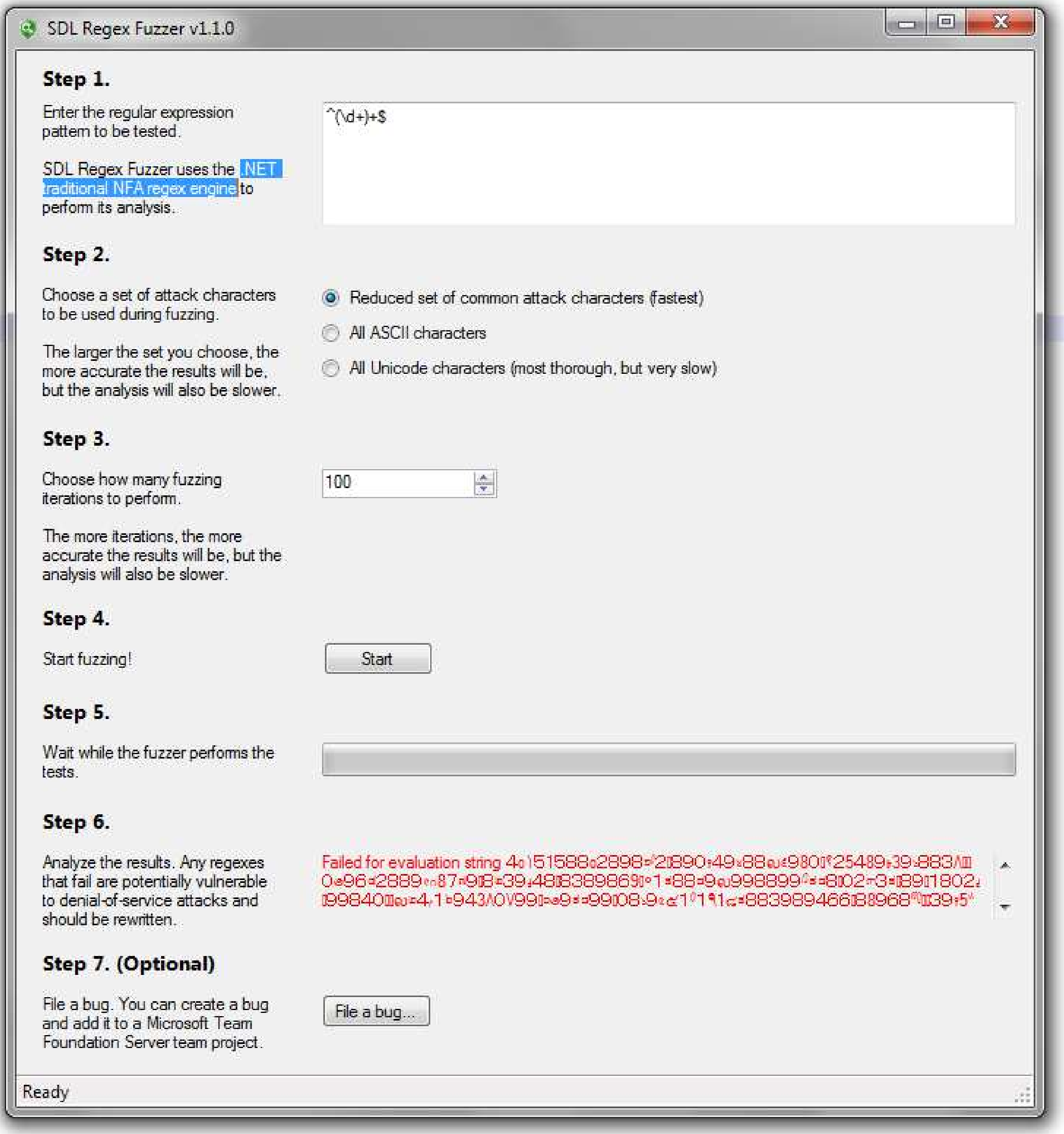
Progress

Fuzzed files: 35 # Failures: 0 main.c

Time	File	Crash

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+)+$



SDL Regex Fuzzer v1.1.0

Step 1.

Enter the regular expression pattern to be tested.

SDL Regex Fuzzer uses the .NET traditional NFA regex engine to perform its analysis.

```
^(\d+)+$
```

Step 2.

Choose a set of attack characters to be used during fuzzing.

The larger the set you choose, the more accurate the results will be, but the analysis will also be slower.

● Reduced set of common attack characters (fastest)

○ All ASCII characters

○ All Unicode characters (most thorough, but very slow)

Step 3.

Choose how many fuzzing iterations to perform.

The more iterations, the more accurate the results will be, but the analysis will also be slower.

100

Step 4.

Start fuzzing!

Step 5.

Wait while the fuzzer performs the tests.

Step 6.

Analyze the results. Any regexes that fail are potentially vulnerable to denial-of-service attacks and should be rewritten.

```
Failed for evaluation string 4 151588 2898 2 890 49 88 980 25489 39 883
0 96 2889 87 9 8 39 48 838986 1 88 9 998899 8 02 3 89 1802
99840 4 1 943 0 99 9 99 08 9 1 1 1 88398946688968 39 5
```

Step 7. (Optional)

File a bug. You can create a bug and add it to a Microsoft Team Foundation Server team project.

Ready

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/

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

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_Decompilers

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

Conclusions

- Annotations for even better results
  - Microsoft SAL
- Dynamic analyzers can profile application
  - and find bugs not found by static analysis
- Fuzzing is blackbox dynamic analysis via malformed inputs

Questions