IBM Global Services

July 2007

Breaking C++ Applications

BlackHat Briefings 2007

Mark Dowd, Neel Mehta,
John McDonald
IBM ISS X-Force R&D

IBM Internet Security Systems
Ahead of the threat.™

Introduction

- **The vast majority of today's commercial applications are developed in C/C++, or have some C/C++ components**
  - Firewalls
  - Application Client/Servers (eg. Web Browsers/servers)
  - Kernels / Kernel Drivers
- **Analyzing such programs for vulnerabilities is a topic that is talked about often within the security and developer communities**
  - Speeches
  - Books
  - Commercial and University courses/certifications
- **Despite this widespread focus, nearly all of the discussion on application vulnerabilities discusses C-centric issues**
  - Unsafe String functions
  - Pointer Arithmetic and other memory management issues
  - Integer problems
- **While this is relevant for C++ applications, there is very little material that deals potential problems for C++ specific problems!**

What We Will Cover….

- **Choosing Your Compiler**
  - The compiler you use to create a binary makes a pretty big difference to how secure your application will be when dealing with certain exceptional circumstances
- **Class Behavior**
  - Classes that don't behave as expected due to awkward design can lead to problems
  - Constructor implementation
  - Operator overloading
  - State inconsistencies
- **Variable Length Arrays**
  - Delete vs delete [] and their implications
- **Exception Handling**
  - Exception Filtering carelessness
  - Exception handling as an exploitation facilitator
  - Special case: Stack exceptions
  - Automation and auditing aids for exception handling

Choosing your compiler

- **Compilers are generally not considered to have security implications**
  - Same code compiled in different build environments should theoretically act and behave roughly the same
  - In reality this is far from the truth
  - This includes not just how the code is compiled but the standard libraries available for different compilers
- **Visual Studio contains additional security features**
  - Stack cookies placed in functions with stack buffers are used to prevent exploitation of buffer overflows (introduced in Visual Studio 2002)
  - Exception handler records also have undergone several iterations of validation routines
  - Fixes for several language-level problems – primarily potential integer overflows when allocating arrays of objects with the new [] operator (fixed in Visual Studio 2005)
  - Changed the way exception handling works in Visual Studio 2005 (more on that later…)
- **Choice of compiler will vastly affect how exploitable vulnerabilities are**
  - Visual Studio 2005 provides the most pro-actively secure builds
  - Many commercial companies lag behind in most current compiler environments

Choosing your compiler

- **The standard libraries you use are also important**
  - Variations in APIs is a topic touched on from time to time for C APIs (snprintf on UNIX vs _snprintf on Windows, etc)
  - Usually only one version of each library for each system component
  - Will change with increased use of “side-by-side” assemblies and “Isolated applications”
  - STL in C++ however is static
  - Whichever STL implementation you used at build time will dictate which version the application uses
  - Compiles code into the target executable
  - STL implementations vary a lot in code quality and robustness
- **STL Implementation fixes/changes**
  - Likewise, STL has fixes performed on it periodically
  - Many different STL implementations are available, usually tied to a compiler
  - Microsoft has the default compiler-available STL’s, as well as separately downloadable “Platform SDK” packages, and more recently “Windows SDK” packages
  - They have fixed a lot of things pretty quietly ;)
  - Latest one (Windows SDK v6) is tough (jerks)
  - **Many commercial applications are compiled with old and buggy STL implementations!**

Example 1 – Platform SDK 2003 R2 (vector class)

```
void insert(iterator _P, size_type _M, const _Ty& _X)
{
       if (_End - _Last < _M)
       {
              size_type _N = size() + (_M < size() ? size() : _M);
              iterator _S = allocator.allocate(_N, (void *)0);
              iterator _Q = _Ucopy(_First, _P, _S);
              _Ufill(_Q, _M, _X);
              _Ucopy(_P, _Last, _Q + _M);

              … more code ...
```

Example 1 – Platform SDK 2003 R2 (vector class)

```
void insert(iterator _P, size_type _M, const _Ty& _X)
{
        if (_End - _Last < _M)
        {
                size_type _N = size() + (_M < size() ? size() : _M);
                iterator _S = allocator.allocate(_N, (void *)0);
                iterator _Q = _Ucopy(_First, _P, _S);
                _Ufill(_Q, _M, _X);
                _Ucopy(_P, _Last, _Q + _M);

                … more code ...
```

Example 1 – Windows SDK Version (vector class)

```
void insert(iterator _Where, size_type _Count, const _Ty& _Val)
{// insert _Count * _Val at _Where
      Insert_n(_Where, _Count, _Val);
}



void _Insert_n(iterator _Where,
       size_type _Count, const _Ty& _Val)
{      // insert _Count * _Val at _Where

       _Ty _Tmp = _Val;     // in case _Val is in sequence
       size_type _Capacity = capacity();

       if (_Count == 0)
              ;
       else if (max_size() - size() < _Count)
              Xlen();// result too long

       … more code ...
```

Example 1 – Windows SDK Version (vector class)

```
void insert(iterator _Where, size_type _Count, const _Ty& _Val)
{// insert _Count * _Val at _Where
      Insert_n(_Where, _Count, _Val);
}



void _Insert_n(iterator _Where,
        size_type _Count, const _Ty& _Val)
{       // insert _Count * _Val at _Where

        _Ty _Tmp = _Val;      // in case _Val is in sequence
        size_type _Capacity = capacity();

        if (_Count == 0)
                ;
        else if (max_size() - size() < _Count)
                Xlen();// result too long

        … more code ...
```

Example 2 – Platform SDK 2003 R2 (allocator routines)

```
            // TEMPLATE FUNCTION _Allocate
template<class _Ty> inline
      _Ty _FARQ *_Allocate(_PDFT _N, _Ty _FARQ *)
      {
            if (_N < 0)
                  _N = 0;
            return ((_Ty _FARQ *)operator new((_SIZT)_N * sizeof (_Ty)));
      }
```

Example 2 – Platform SDK 2003 R2 (allocator routines)

```
            // TEMPLATE FUNCTION _Allocate
template<class _Ty> inline
      _Ty _FARQ *_Allocate(_PDFT _N, _Ty _FARQ *)
      {
            if (_N < 0)
                  _N = 0;
            return ((_Ty _FARQ *)operator new((_SIZT)_N * sizeof (_Ty)));
      }
```

Example 2 – Windows SDK Version (allocator routines)

```
          // TEMPLATE FUNCTION _Allocate
template<class _Ty> inline
     _Ty _FARQ *_Allocate(_SIZT _Count, _Ty _FARQ *)
     {    // check for integer overflow
          if (_Count <= 0)
               _Count = 0;
          else if (((_SIZT)(-1) / _Count) < sizeof (_Ty))
               _THROW_NCEE(std::bad_alloc, NULL);

          // allocate storage for _Count elements of type _Ty
          return ((_Ty _FARQ *)::operator new(_Count * sizeof (_Ty)));
     }
```

Example 2 – Windows SDK Version (allocator routines)

```
              // TEMPLATE FUNCTION _Allocate
template<class _Ty> inline
       _Ty _FARQ *_Allocate(_SIZT _Count, _Ty _FARQ *)
       {      // check for integer overflow
              if (_Count <= 0)
                     _Count = 0;
              else if (((_SIZT)(-1) / _Count) < sizeof (_Ty))
                     _THROW_NCEE(std::bad_alloc, NULL);

              // allocate storage for _Count elements of type _Ty
              return ((_Ty _FARQ *)::operator new(_Count * sizeof (_Ty)));
       }
```

Which STL / Compiler was this binary built with?

- **Knowing compiler version is important, so that protection mechanisms could be identified**
  - Presence of stack cookies (> VS6)
  - __CxxFrameHandler( ) compiler signature (we will look at this later – but can distinguish compilers VS6 / 2002-3 / 2005)
  - Runtime library usage
    - MSVCR80.DLL - VS2005
    - MSVCR71.DLL – VS2003 SP1
    - MSVCR70.DLL – VS2003 SP0
    - MSVCRT.DLL – VS6
  - Note that MSVCRT.DLL is the system CRT, and every compiler can link against it instead of their versioned CRT (with the exception of VS2005, according to MS docs)
  - Windows binaries often use the MSVCRT.DLL
  - Analyzing compiled code itself would give away large clues too, but it's beyond the scope of this talk

Which STL / Compiler was this binary built with?

- **STL version is also critical, due to aforementioned bugs**
  - Again, versions somewhat given away by runtime libraries (MSVCP*.DLL/MSVCR*.DLL)
  - Code for many of the classes is compiled into the binary directly (example: vectors) using the source from the Platform/Windows SDK source code
  - Need to analyze the compiled code
  - Look at some of the example bugs we presented, or implementation changes where a small block of code has been replaced by a function call
  - One big giveaway: use of overflow checking preceding calls to new (using the ‘seto’ instruction)
  - Alternatively: Bindiff against your own compiled version of a function! (Thanks, Halvar)

Class Behavior

- **C++ classes are composed of attributes and methods**
  - Generally, good programming practice guides recommend hiding of attributes, and exposing range of functionality via methods
  - Allows for implementation changes within the class itself to be easy to do without modifying an external interface
- **Methods must maintain the consistency of the instantiated object at all times**
  - Modifying method attributes needs to be done such that the object is left in a usable state
  - If an error occurs in a member function, how is it signaled?
  - Are members left unchanged that need to be changed? Are members changed that need to be reverted in the case of an error?
  - What are the implications of further method invocations on that object after an error has occurred? Especially if it has not been handled properly?

Class Behavior - Constructors

- **Constructors are required to initialize an object into a consistent and usable state**
  - Constructors intended to initialize the object
  - Do they forget to initialize anything?
  - They can fail like anything else, the most likely case being due to a memory allocation error
- **Constructors usually don’t return a value**
  - If an initialization function fails, there is no error code it can return
  - They can throw an exception, but often developers neglect to do this
  - Constructor inconsistencies can lead to accessing an object that hasn’t been correctly setup
  - Most likely outcomes: Uninitialized variable usage, or (less interestingly) NULL-dereferences

Class Behavior

```
Board::Board(unsigned int x_dim, unsigned int y_dim) {
       int len = x_dim * y_dim;
       m_x = 0;
       m_y = 0;
       m_squares = NULL;

       if (x_dim > 0 && (x_dim <= (unsigned int)-1/y_dim)
              && len  <= (size_t)-1/sizeof(m_squares[0])) {
              m_x = x_dim;
              m_y = y_dim;
              m_squares = (int*)calloc(len, sizeof(m_squares[0]));
       }
}

// returns square at a given coordinate
int Board::getSquare(unsigned int x, unsigned int y) {
       if (x >= m_x || y >= m_y) return Board::ERROR_SQUARE;
       return m_squares[y * m_x + x];
}

// sets square at a given x-ycoordinate
bool Board::setSquare(unsigned int x, unsigned int y, int square) {
       if (x >= m_x || y >= m_y) return false;
       m_squares[y * m_x + x] = square;
       return true;
}
```

Class Behavior

```
Board::Board(unsigned int x_dim, unsigned int y_dim) {
       int len = x_dim * y_dim;
       m_x = 0;
       m_y = 0;
       m_squares = NULL;

       if (x_dim > 0 && (x_dim <= (unsigned int)-1/y_dim)
              && len  <= (size_t)-1/sizeof(m_squares[0])) {
              m_x = x_dim;
              m_y = y_dim;
              m_squares = (int*)calloc(len, sizeof(m_squares[0]));
       }
}

// returns square at a given coordinate
int Board::getSquare(unsigned int x, unsigned int y) {
       if (x >= m_x || y >= m_y) return Board::ERROR_SQUARE;
       return m_squares[y * m_x + x];
}

// sets square at a given x-ycoordinate
bool Board::setSquare(unsigned int x, unsigned int y, int square) {
       if (x >= m_x || y >= m_y) return false;
       m_squares[y * m_x + x] = square;
       return true;
}
```

Class Behavior - Destructors

- **Destructor methods need to also be examined carefully**
  - Destructors usually make the assumption that the object is in some sort of consistent state
  - Sometimes if member functions fail partway through, this is not the case
  - Leads to double free's, memsetting out of bounds, etc
- **For stack objects, destructors are called implicitly on function exit**
  - Any time an object has been created on the stack and a function later returns, the destructor is called implicitly
  - In VS, this is typically done by the __CxxFrameHandler function, which uses a lookup table of sorts to see what objects it needs to destruct
  - In g++, this is done by a small section of code compiled into the function right before the function epilogue. For exception handling, there is usually an additional block of code adjacent to the function that is used by the unwinding code. This block of code will end with a call to _Unwind_Resume()
  - Objects are also auto-destructed when an exception occurs
  - Similarly, global objects are destructed implicitly when a program exit occurs
  - We will look at EH later on..

Class Behavior – Overloading Operators

- **Operator overloading in C++ can also have consequences**
  - Operators are expected to have specific semantics, but overloading operators can change this expected behavior
- **new Operator overloading**
  - Prime example of semantic changing that can lead to unintuitive behavior
  - Returning NULL instead of throwing an exception can easily lead to NULL dereferences or worse
  - Note that implementations of “new” vary – some throw an exception, some return NULL, so expected / portable behavior is difficult anyway
  - Overloading new[] is similarly afflicted
  - Overloading new[] will not disable the integer overflow check in VS2005, however custom allocation routines (especially involving adding a header/rounding up) are very often the cause integer overflows
  - Developers should be aware that in the case of integer overflow, the size parameter passed to new[] will be 0xFFFFFFFF

Class Behavior – Auditing Classes

- **Three step process**
  - Enumerate internal state
    - Make note of each piece of internal state
      - member variables and member objects
    - References to external variables and functions
    - Inherited state from parent classes
  - Determine responsibility
    - For members that have associated memory
    - Who is responsible for allocation of the memory
    - Who is responsible for deallocation
    - Where does this happen
  - Determine invariants
    - Relationships between member variables
    - Relationships that should hold true throughout lifetime of class

Class Behavior – Auditing Classes

Example Class Audit Log

```
class buffer
{
    void *memory_ptr;
    size_t buffer_size;
    size_t write_ptr;
    ...
```

Name	Type	Responsibilities	Invariants
memory_ptr	void *	Initialized in constructor Free in destructor Reallocated in resize()	Must point to valid memory Must not be NULL
buffer_size	size_t	Set in constructor Modified in resize()	Must track length of memory at memory_ptr
write_ptr	size_t	Set in constructor Modified in write() Modified in resize()	Must be between 0 and buffer_size-1

Class Behavior – Auditing Construction/Destruction

- **Auditing constructors**
  - Match initialization of variables with destruction of variables from destructors
  - Check for memory allocations or access to system resources (such as files etc). Can they fail?
  - What happens in the case of an error? Be on the lookout for “nothing”
  - Not necessarily a bug, you need to see how subsequent function calls will handle it
  - If they access member functions under the assumption they’re correctly initialized, then you have a problem
- **Auditing destructors**
  - Make a note of any resources they free up
  - Memory accesses particularly interesting
  - In particularly look for destruction of member variables that are destroyed elsewhere in an objects lifetime – if the other place can possibly generate an exception or error, can de-allocation happen twice?
  - Remember: Exceptions can happen anywhere, and for stack-based objects this means automatic destruction

Class Behavior – Auditing Class Behavior

- **Auditing operators**
  - Note any operator overloads where behavior deviates from standard operator behavior
  - Eg, new returning NULL, operator+ or operator+= not throwing an exception
  - Think about possible corner cases – what happens if operator= is used to assign an object to itself?
  - These add up to vulnerabilities if objects use operators incorrectly or errors go unnoticed
- **Auditing member functions**
  - When auditing a member function, leaving object inconsistent is a primary issue
  - Check whether objects might be left in a potentially dangerous state
  - Achieved by marking all the points where the function exits, and what the object state can be at those points
  - Remember: this includes exceptions being thrown!
  - Exceptions from sub-functions are more likely to cause problems that ones thrown from the member function itself
  - Allocation-based exceptions can probably be induced in many cases and barely no-one handles it
  - Triggering a stack unwind at an unlikely juncture for a memory allocation could be interesting

Variable Length Arrays

- **C++ allows for variable length array allocations using the new [] operator**
  - VLAs are an alternative to container classes (vectors, etc)
  - Look and feel of C arrays (subscripting etc)
  - Unlike container classes, no access protection is guaranteed
  - Dynamic memory is managed using new [] / delete [] as opposed to new and delete functions reserved for scalar constructions / destructions
  - New and new [] function differently, as do delete and delete []
  - Mixing the vector / scalar new and delete operators produce "undefined" results

Variable Length Arrays

```
bool bob(char *string)
{
       MyClass *ba = new MyClass[10];
       bool rc;

       rc = act_on_objects(ba);

       if(rc == false)
       {
              delete ba;
              return false;
       }

       ... more code ...
}
```

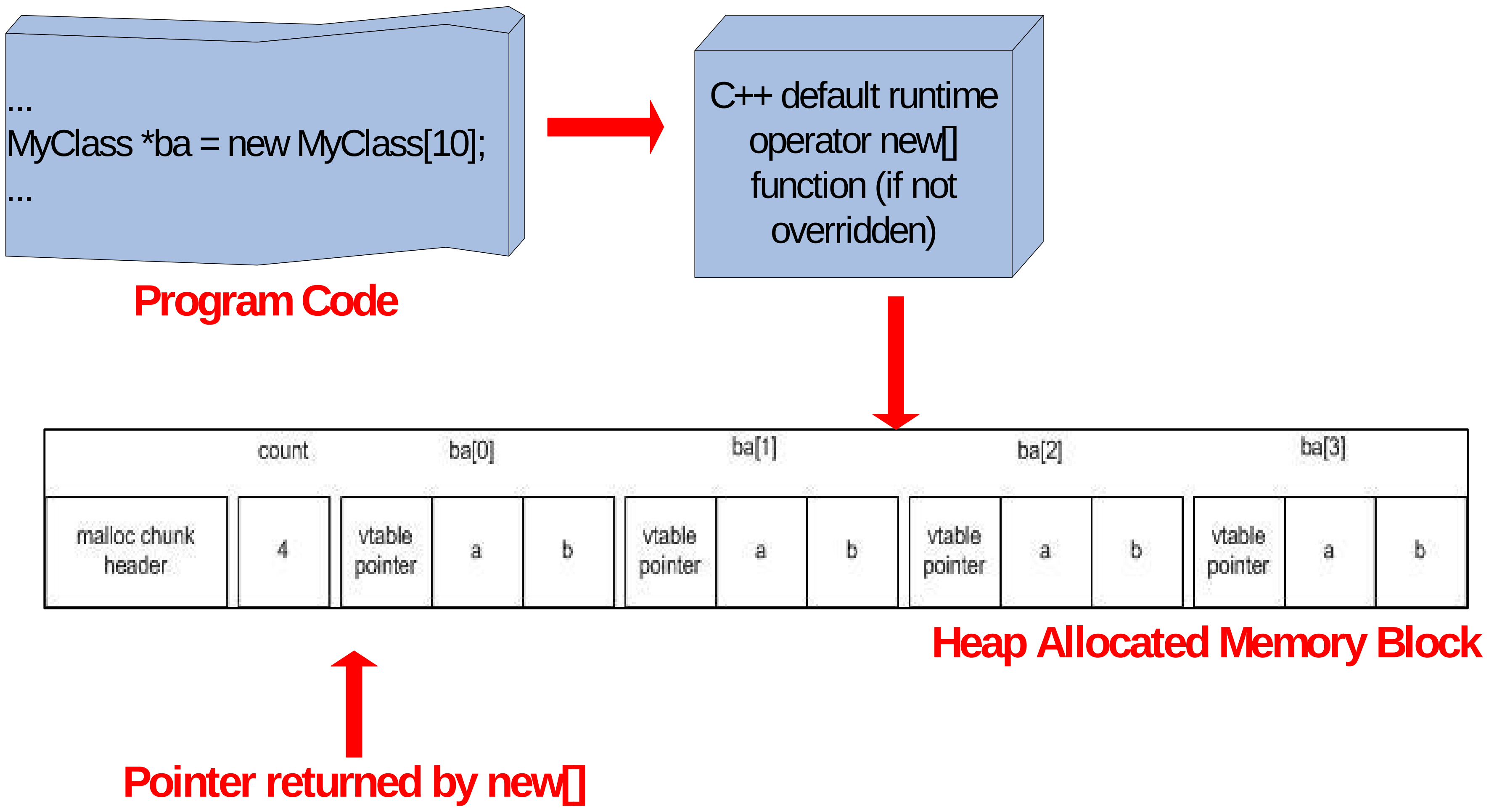
Variable Length Arrays

```
bool bob(char *string)
{
        MyClass *ba = new MyClass[10];
        bool rc;

        rc = act_on_objects(ba);

        if(rc == false)
        {
                delete ba;
                return false;
        }

        ... more code ...
}
```

Variable Length Arrays – Mixing

- **Vectors vs Scalars**
  - Scalars are singular dynamically allocated objects
  - The new operator simply allocates an appropriately sized block, and calls the constructor for the relevant object type
  - Arrays are allocated as a contiguous block of memory – objects are aligned one after the other in sequence with a DWORD preceding the objects that indicate how many members there are
  - This is required so when delete [] is called, it knows how many objects to destruct
  - Pointer returned by new [] is a pointer to the first object in the array – the DWORD count preceding the objects is “invisible” to the caller
  - Not the case for primitive types

Array Allocation in C++


...
MyClass *ba = new MyClass[10];
...
Program Code
C++ default runtime operator new[] function (if not overridden)
count
ba[0]
ba[1]
ba[2]
ba[3]
malloc chunk header
4
vtable pointer
a
b
vtable pointer
a
b
vtable pointer
a
b
vtable pointer
a
b
Heap Allocated Memory Block
Pointer returned by new[]

Variable Length Arrays – Mixing

- **Using delete on an array**
  - Only the first object will be deleted, resulting in memory leaks
  - More subtle problem: a pointer will be passed to free() that was never returned by malloc(), since the pointer returned by new [] is (allocated_array + sizeof(DWORD))
  - Cases where RtlFreeHeap() is called are not interesting, because the pointer won't be 8-byte aligned, which causes RtlFreeHeap() to do nothing
  - In Windows, it depends on which CRT library is in use –generally, MSVCRT vulnerable, MSVCR*.DLL not vulnerable in most installs
  - Due to the use of the __sbh_alloc() allocation function in MSVCRT (pre-Vista)
  - Depends on __active_heap parameter, which defaults to __SYSTEM_HEAP for XP (you want __V5_HEAP or __V6_HEAP)
- **Using delete [] on a scalar object**
  - Much less likely
  - Consequences would be immediately noticeable in most cases, unless the code path was very rare
  - More dangerous – 4 bytes preceding allocated memory chunk (ie. Part of the chunk header) is treated as an element count DWORD
  - Destructors called on memory locations past the end of the allocated object
  - G++: If vtables are used in destruction, arbitrary memory locations called
  - Not the case for MS binaries – they only use the vtable from the 0-offset element
  - Both of these issues dealt with in-depth in a blog article we posted in January (http://taossa.com/index.php/2007/01/03/attacking-delete-and-delete-in-c/#more-52)

Variable Length Arrays - Auditing

- **Auditing for improper delete**
  - Make a note of all variables that are allocated using the new [] operator
  - Trace use of the variable until deallocation
  - Make sure delete [] is used in all control flow paths
  - Any use of delete is suspect
  - This lends itself towards simple automated analysis using regular expressions

Exception Handling

- **Exception Handling is one of the major features of C++ (and indeed most OO-languages)**
  - Used by standard libraries to indicate system errors
  - Custom exceptions thrown by many applications
- **Generally EH is considered to be a device that improves robustness (and therefore security) by providing developers the ability to signal error conditions “out-of-band”**
  - Allows for extended information about error conditions to be encapsulated
  - Usually forces a programmer to handle a particular type of error
- **Exploration of exception handling and its implications on security requires us to be familiar with how exception handling is implemented at the compiler/system level**

Windows Exception Handling

- **Windows uses its native Structured Exception Handling (SEH) functionality to implement C++ EH**
  - Native Exception handling functionality provides a basic framework for exceptions to be caught, examined and handled on a per-thread basis
  - Used to implement the __try { } / __except { } / __finally { } keywords
  - Excellent resources for the internals of SEH are Matt Pietrek's paper "A Crash Course on the Depths of Win32 Structured Exception Handling" (http://www.microsoft.com/msj/0197/exception/exception.aspx) and Igorsk's article "Reversing Microsoft Visual C++ Part I: Exception Handling" (http://www.openrce.org/articles/full_view/21).
- **SEH records are a simple 2 member structure**
  - Previous handler pointer – points to the previous exception record, or 0xFFFFFFFF if there is no previous record
  - Pointer to dispatch routine which returns ExceptionContinueExecution (0) or ExceptionContinueSearch (1). (Can also return ExceptionNestedException or ExceptionCollidedUnwind.)

Windows Exception Handling (A little background…)

- **Implementation of __try { } / __except { } / __finally { } adds an additional layer of complexity**
  - A single SEH record is placed in the relevant functions stack frame
  - It calls a standardized function which determines whether the exception should be caught and execution should restart at the faulting address or be transferred to a handler
  - These functions have gone through several revisions over the years, but they essentially achieve the same goal: deciding whether the exception should be handled, and if so, where control should be transferred to
  - Standardized function depends on compiler – often you will see _except_handler3, or _except_handler4 if compiled with Visual Studion 2005 or above
  - Both achieve the same thing, although _except_handler4 has additional security features
  - MS Compiler intrinsically adds this exception management code in
- **Scope Tables**
  - Both SEH3 and SEH4 functions take a pointer to a structure as an argument, which contains a scope table
  - Scope table records consist of a filter function pointer, a handler function pointer, and an enclosing try level
  - Scope table search starts at an index given by the try level variable in the functions stack frame

Windows Exception Handling (A little background…)

- **Handler and Filter Functions**
  - Filter function pointer may be NULL
  - If present, it is executed. Return value > 0 means handle the exception (execute corresponding handler), 0 means don’t handle the exception (continue search), and < 0 means restart execution at faulting instruction
  - If filter function isn’t present, the handler function will be executed unconditionally when this exception frame is unwound
  - If filter function returns 0, the enclosing try level is used as the next index in the scope table to check. A value of -1 (or -2 for SEH4) means the search is over and nothing was found
- **Exception Handling is a 2-pass process**
  - First phase iterates through exception handler records trying to find one that will handle the exception
  - Second phase is the “unwinding” phase, where exception records that declined to handle the exception have the opportunity to perform any cleanup that they need to do

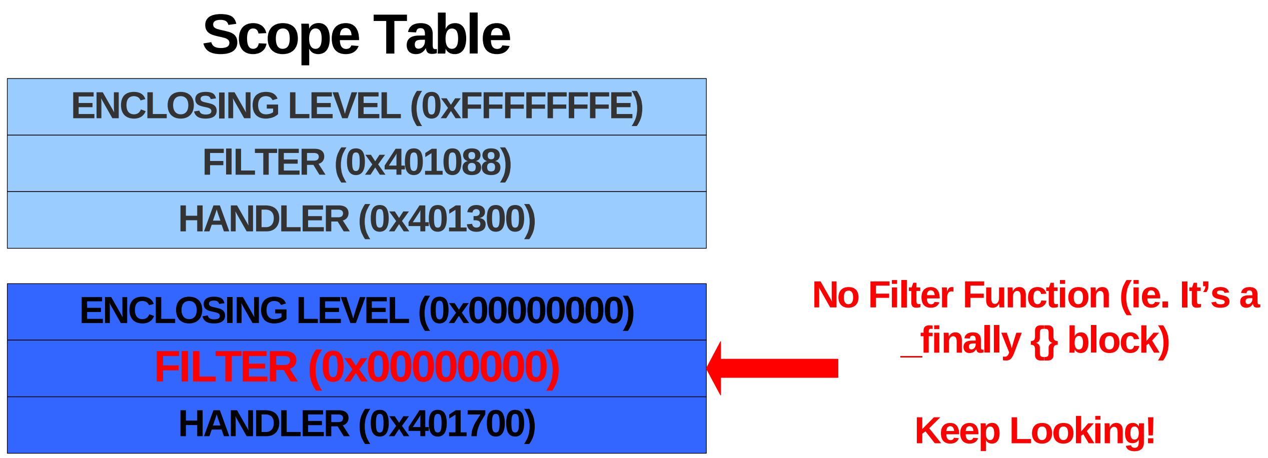
Handling an Exception – The Scope Table

Scope Table

ENCLOSING LEVEL (0xFFFFFFFE)
FILTER (0x401088)
HANDLER (0x401300)

ENCLOSING LEVEL (0x00000000)
FILTER (0x00000000)
HANDLER (0x401700)

Handling an Exception – Pass 1



Scope Table

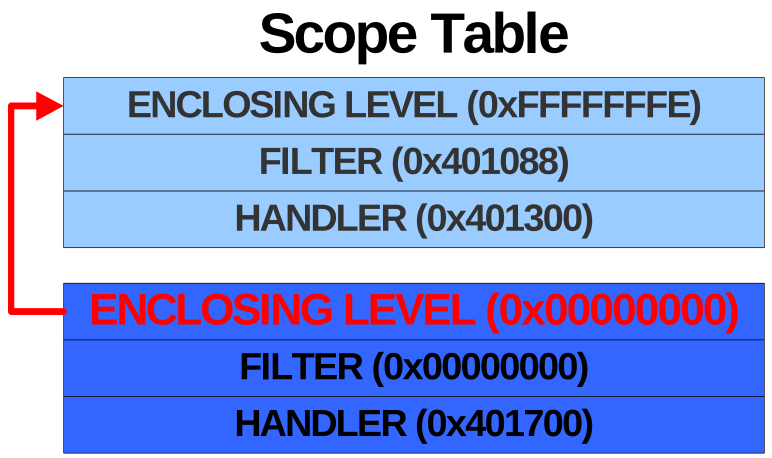
ENCLOSING LEVEL (0xFFFFFFFE)
FILTER (0x401088)
HANDLER (0x401300)

ENCLOSING LEVEL (0x00000000)
FILTER (0x00000000)
HANDLER (0x401700)

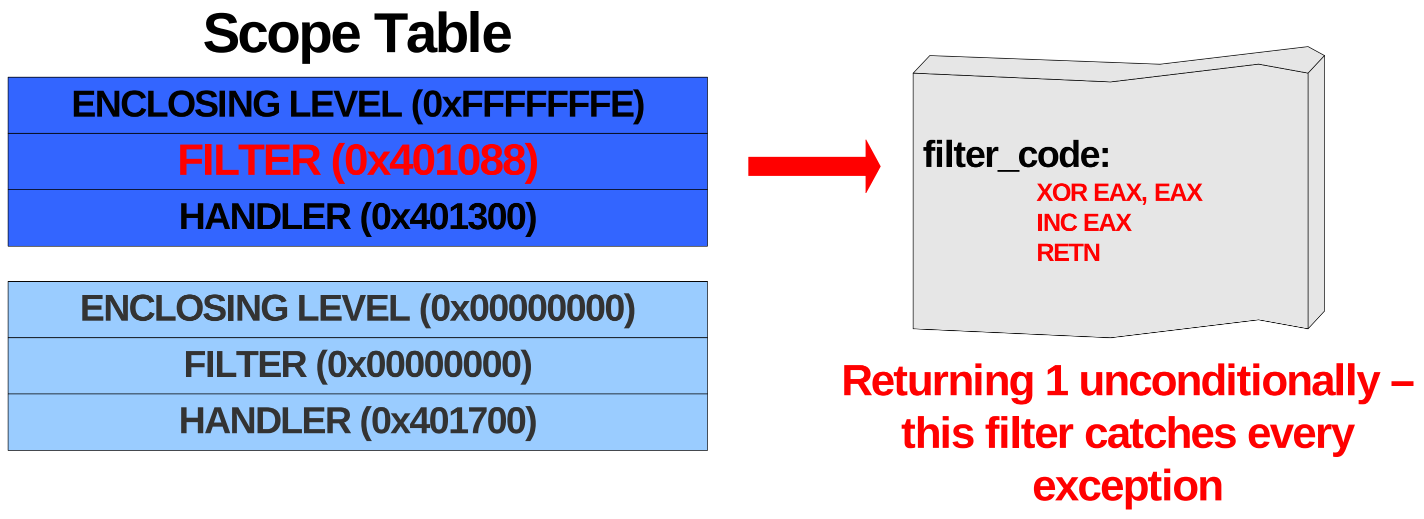
No Filter Function (ie. It's a _finally {} block)

Keep Looking!

Handling an Exception – Pass 1



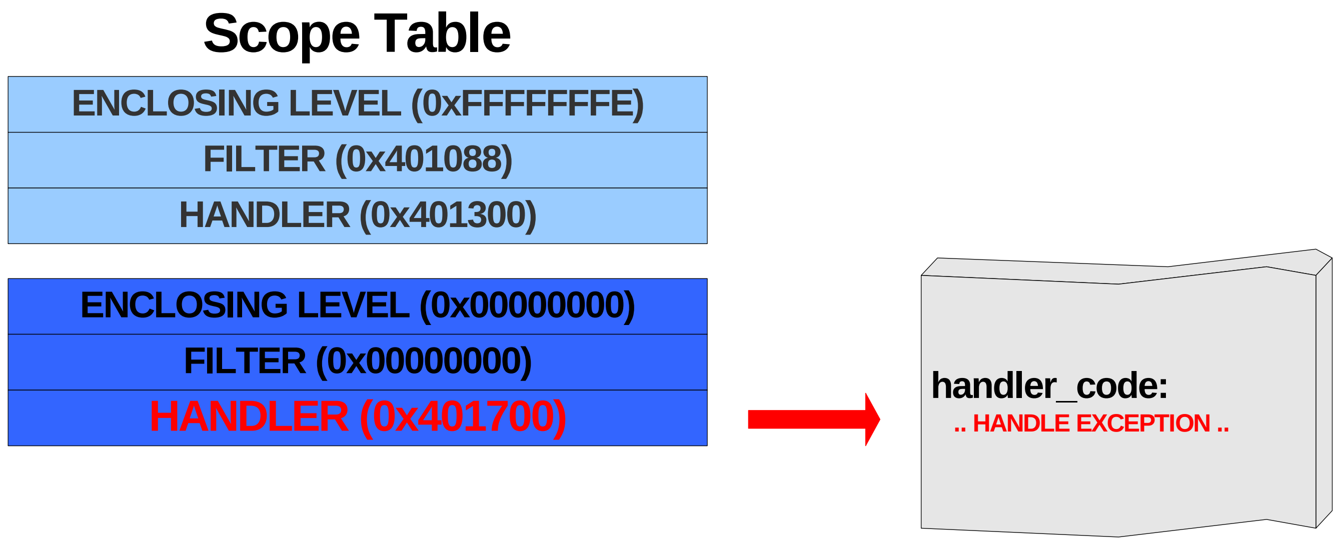
Handling an Exception – Pass 1



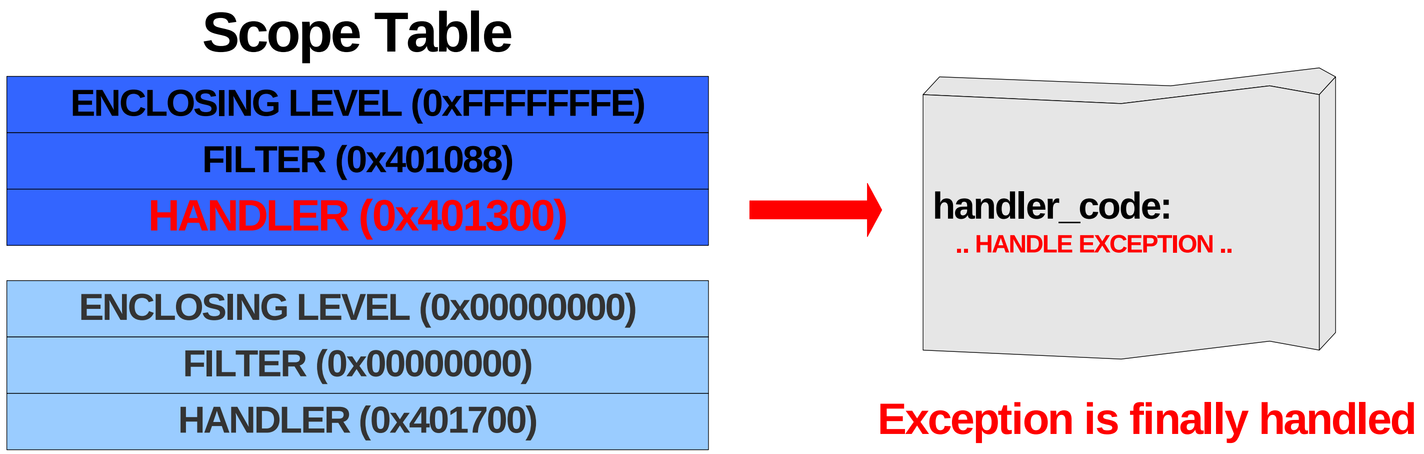
```
filter_code:
          XOR EAX, EAX
          INC EAX
          RETN
```

Returning 1 unconditionally – this filter catches every exception

Handling an Exception – Pass 2



Handling an Exception – Pass 2



Windows Exception Handling

- **When an exception of any kind is generated, quite a lot of code is potentially executed**
  - An exception on Windows isn’t necessarily fatal, and even if it is, a lot of things potentially happen before the program dies
  - Since abrupt termination isn’t immediate, we need to pay careful attention to what actually does happen
  - Caught Exceptions are also interesting – what are the implications of a usually fatal exception being caught?
  - Exception blocks often go largely ignored in code audits, and are often missed during QA
  - When auditing binaries, usually this is because the code paths are not immediately obvious
- **Exceptions that aren’t handled correctly are also potentially problematic**
  - Locks often need to be released
  - Memory might need to be released
  - Global variables might need to be corrected
- **Ability for attackers to generate exceptions is usually somewhat limited**

Windows System Exceptions (from WinBase.h)

```
#define WAIT_IO_COMPLETION                    STATUS_USER_APC
#define STILL_ACTIVE                          STATUS_PENDING
#define EXCEPTION_ACCESS_VIOLATION            STATUS_ACCESS_VIOLATION
#define EXCEPTION_DATATYPE_MISALIGNMENT       STATUS_DATATYPE_MISALIGNMENT
#define EXCEPTION_BREAKPOINT                  STATUS_BREAKPOINT
#define EXCEPTION_SINGLE_STEP                 STATUS_SINGLE_STEP
#define EXCEPTION_ARRAY_BOUNDS_EXCEEDED       STATUS_ARRAY_BOUNDS_EXCEEDED
#define EXCEPTION_FLT_DENORMAL_OPERAND        STATUS_FLOAT_DENORMAL_OPERAND
#define EXCEPTION_FLT_DIVIDE_BY_ZERO          STATUS_FLOAT_DIVIDE_BY_ZERO
#define EXCEPTION_FLT_INEXACT_RESULT          STATUS_FLOAT_INEXACT_RESULT
#define EXCEPTION_FLT_INVALID_OPERATION       STATUS_FLOAT_INVALID_OPERATION
#define EXCEPTION_FLT_OVERFLOW                STATUS_FLOAT_OVERFLOW
#define EXCEPTION_FLT_STACK_CHECK             STATUS_FLOAT_STACK_CHECK
#define EXCEPTION_FLT_UNDERFLOW               STATUS_FLOAT_UNDERFLOW
#define EXCEPTION_INT_DIVIDE_BY_ZERO          STATUS_INTEGER_DIVIDE_BY_ZERO
#define EXCEPTION_INT_OVERFLOW                STATUS_INTEGER_OVERFLOW
#define EXCEPTION_PRIV_INSTRUCTION            STATUS_PRIVILEGED_INSTRUCTION
#define EXCEPTION_IN_PAGE_ERROR               STATUS_IN_PAGE_ERROR
#define EXCEPTION_ILLEGAL_INSTRUCTION         STATUS_ILLEGAL_INSTRUCTION
#define EXCEPTION_NONCONTINUABLE_EXCEPTION    STATUS_NONCONTINUABLE_EXCEPTION
#define EXCEPTION_STACK_OVERFLOW              STATUS_STACK_OVERFLOW
#define EXCEPTION_INVALID_DISPOSITION         STATUS_INVALID_DISPOSITION
#define EXCEPTION_GUARD_PAGE                  STATUS_GUARD_PAGE_VIOLATION
#define EXCEPTION_INVALID_HANDLE              STATUS_INVALID_HANDLE
#define EXCEPTION_POSSIBLE_DEADLOCK           STATUS_POSSIBLE_DEADLOCK
#define CONTROL_C_EXIT                        STATUS_CONTROL_C_EXIT
```

Windows System Exceptions (from WinBase.h)

```
#define WAIT_IO_COMPLETION                    STATUS_USER_APC
#define STILL_ACTIVE                          STATUS_PENDING
#define EXCEPTION_ACCESS_VIOLATION            STATUS_ACCESS_VIOLATION
#define EXCEPTION_DATATYPE_MISALIGNMENT       STATUS_DATATYPE_MISALIGNMENT
#define EXCEPTION_BREAKPOINT                  STATUS_BREAKPOINT
#define EXCEPTION_SINGLE_STEP                 STATUS_SINGLE_STEP
#define EXCEPTION_ARRAY_BOUNDS_EXCEEDED       STATUS_ARRAY_BOUNDS_EXCEEDED
#define EXCEPTION_FLT_DENORMAL_OPERAND        STATUS_FLOAT_DENORMAL_OPERAND
#define EXCEPTION_FLT_DIVIDE_BY_ZERO          STATUS_FLOAT_DIVIDE_BY_ZERO
#define EXCEPTION_FLT_INEXACT_RESULT          STATUS_FLOAT_INEXACT_RESULT
#define EXCEPTION_FLT_INVALID_OPERATION       STATUS_FLOAT_INVALID_OPERATION
#define EXCEPTION_FLT_OVERFLOW                STATUS_FLOAT_OVERFLOW
#define EXCEPTION_FLT_STACK_CHECK             STATUS_FLOAT_STACK_CHECK
#define EXCEPTION_FLT_UNDERFLOW               STATUS_FLOAT_UNDERFLOW
#define EXCEPTION_INT_DIVIDE_BY_ZERO          STATUS_INTEGER_DIVIDE_BY_ZERO
#define EXCEPTION_INT_OVERFLOW                STATUS_INTEGER_OVERFLOW
#define EXCEPTION_PRIV_INSTRUCTION            STATUS_PRIVILEGED_INSTRUCTION
#define EXCEPTION_IN_PAGE_ERROR               STATUS_IN_PAGE_ERROR
#define EXCEPTION_ILLEGAL_INSTRUCTION         STATUS_ILLEGAL_INSTRUCTION
#define EXCEPTION_NONCONTINUABLE_EXCEPTION    STATUS_NONCONTINUABLE_EXCEPTION
#define EXCEPTION_STACK_OVERFLOW              STATUS_STACK_OVERFLOW
#define EXCEPTION_INVALID_DISPOSITION         STATUS_INVALID_DISPOSITION
#define EXCEPTION_GUARD_PAGE                  STATUS_GUARD_PAGE_VIOLATION
#define EXCEPTION_INVALID_HANDLE              STATUS_INVALID_HANDLE
#define EXCEPTION_POSSIBLE_DEADLOCK           STATUS_POSSIBLE_DEADLOCK
#define CONTROL_C_EXIT                        STATUS_CONTROL_C_EXIT
```

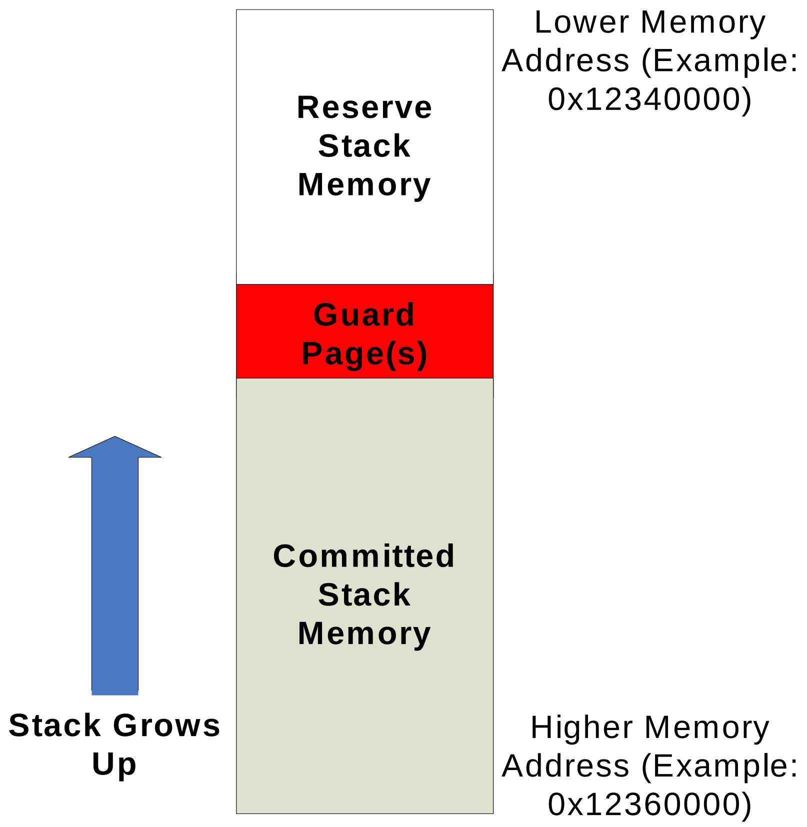
Stack Layout

- **When a thread starts, a chunk of memory is reserved out of the process address space, and a small amount of that memory (typically 1 page) is committed**
  - Stack is allocated during thread initialization
  - Standard thread stacks are divided into 3 sections – the "committed" part of the stack (actively in use memory that is backed by memory pages or swap), a guard page at the end of the committed section (the "soft" guard page), and the reserved (but uncommitted) remainder of the stack
  - Sizes of reserve/commits are found in the executables PE header, but these can be overridden to specify different values in calls to CreateThread().
  - The values for Stack reserve/commit in the PE header of a DLL are ignored in preference for those in the base executable
- **Thread stack will never exceed the reserve size specified**

Guard Pages

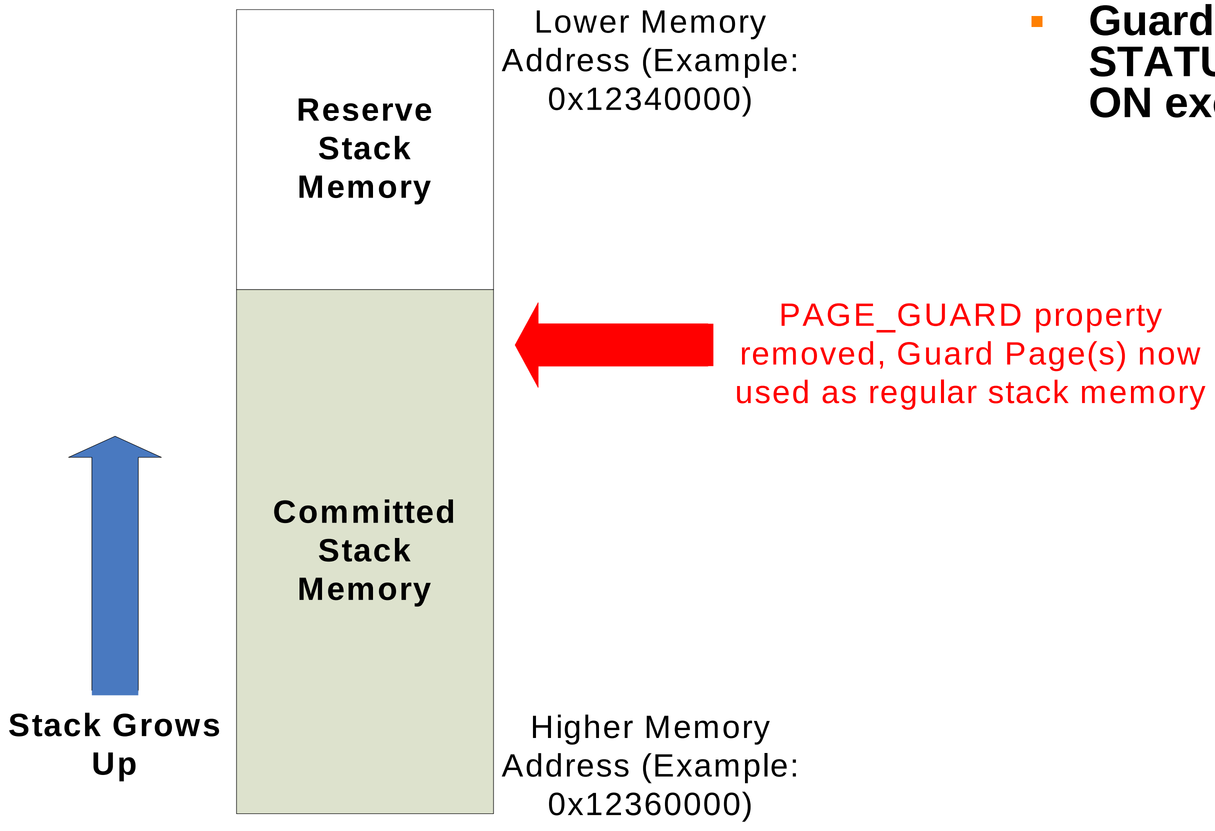
- **Guard Pages are protected using _ZwProtectVirtualMemory with flProtect parameters PAGE_READWRITE | PAGE_GUARD**
  - When the committed region of the stack is allocated, it is also protected with the PAGE_GUARD protection property
  - When the first page is touched, the PAGE_GUARD property is REMOVED from the page and STATUS_GUARD_PAGE_VIOLATION exception is thrown (0x80000001)
  - Usually the kernel will automatically handle this by committing more stack memory and setting up a new guard page higher in memory
  - If the stack cannot grow any further, a STATUS_STACK_OVERFLOW exception is thrown (0xC00000FD)
- **“Hard” Guard Pages vs “Soft” Guard Pages**
  - In addition to the guard page (sometimes known as the “soft” guard page), there is an additional page after the soft guard page known as the “hard” guard page
  - The “hard” guard page will be a page that is reserved but never committed
  - It has permissions PAGE_NOACCESS as well
  - Prevents exception handlers (and possibly other code in misbehaving threads) from eating up stack space and running into adjacent allocated memory regions
  - Hard guard page is not present in cases where stack commit size == stack reserve size (unlikely, but possible)

Stack Dynamic Resizing


Lower Memory Address (Example: 0x12340000)
Reserve Stack Memory
Guard Page(s)
Committed Stack Memory
Stack Grows Up
Higher Memory Address (Example: 0x12360000)

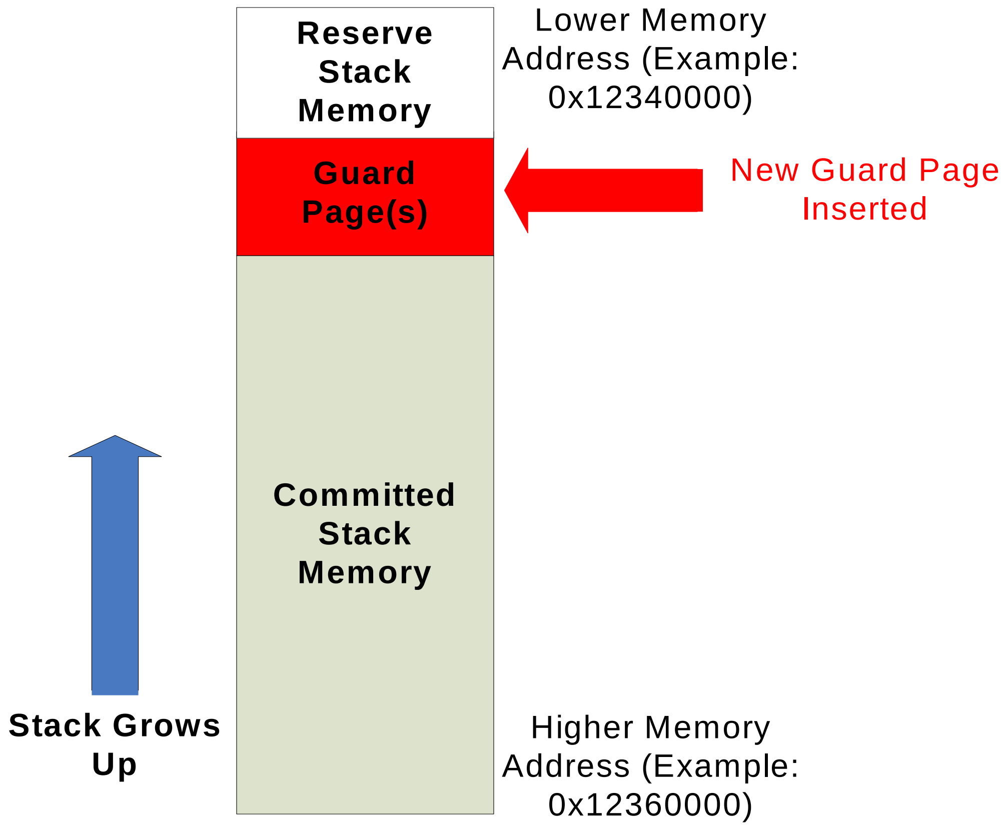
- **Initial Stack Layout – Committed memory is usually only one page**
- **Committed memory is guarded using ZwProtectVirtualMemory()**
- **Reserve memory is uncommitted**
- **Work is performed by RtlpCreateStack() from ntdll.dll or BaseCreateStack() from kernel32.dll, depending on the circumstances**
- **Stack is ready for use**

Stack Dynamic Resizing


Lower Memory Address (Example: 0x12340000)
Reserve Stack Memory
PAGE_GUARD property removed, Guard Page(s) now used as regular stack memory
Committed Stack Memory
Stack Grows Up
Higher Memory Address (Example: 0x12360000)

- **Guard page is touched, STATUS_GUARD_PAGE_VIOLATION exception thrown**

Stack Dynamic Resizing



- **MMAccessFault() in ntoskrnl.exe receives page fault**
- **MiAccessCheck() determines that a guard page was hit**
- **Stack overflow is checked for and handled by calling MiCheckForUserStackOverflow()**
- **If there is more than 1 page left in the stack reserve, it will allocated one new page and make it a guard page**
- **Otherwise, it will return STATUS_STACK_OVERFLOW**
- **Last page remains untouched ("hard" guard page)**

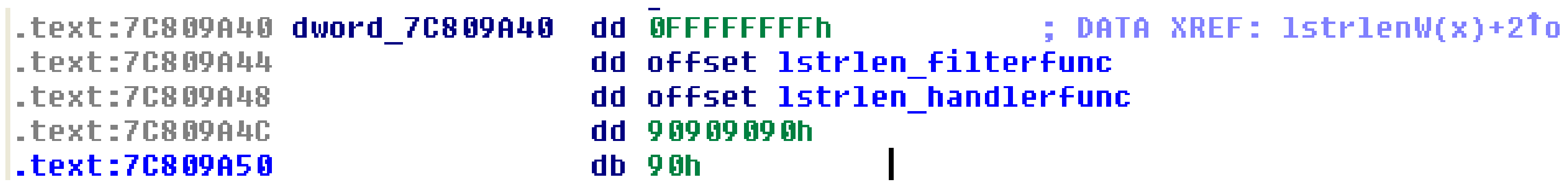
Exploiting Stack Overflow Exceptions

- **Stack overflow exceptions can happen at any time**
  - Conceptually, a stack overflow exception could be generated for any push instruction, call instruction, or mov instruction (if moving a value into a local variable)
  - Due to how SEH_prolog() works, local variables in a function with EH installed cannot generate an overflow exception on access, however called functions can
  - In reality, they don't often happen because the default reserve size for a stack is ~ 1 MB
  - In the case where a variable length alloca() is done or recursion is performed however, a stack exception might be coerced
  - Usually in this case, the faulting thread will just terminate
- **Catch-all Exception handlers to the rescue!**
  - If exception handling is in place, there is quite a bit of work to be done before a thread is terminated (as we have seen)
  - In particularly, poorly protected code that does catch-all exception handling will continue execution
  - The ability to trigger an unexpected exception might have bought us something
  - After the exception is triggered, our stack space is limited but it's not particularly restrictive – we have what was previously the soft guard page to use freely now (so we have 4k)
  - Plenty for execution to continue in many cases

Exploiting Stack Overflow Exceptions – API Examples

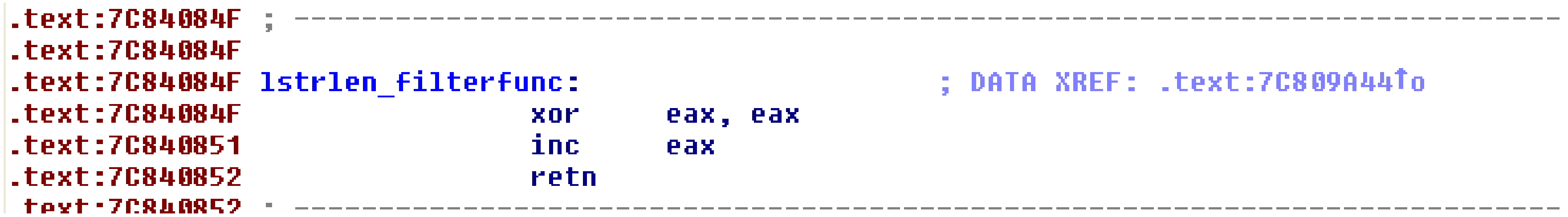
- **The lstrlen() function installs a catch-all exception handler**
  - The lstrlenA() and lstrlenW() functions are intended to take a string and return a length, but also fail quietly without crashing when NULL/invalid pointers are received
  - They achieve this by installing catch-all SEH record and then trying a regular strlen
  - lstrlenA() doesn't use any stack space after installing a SEH record but lstrlenW() does
  - If a stack overflow occurs when calling wcslen(), lstrlenW() will catch it with a catch-all exception handler
  - Default disposition is to return 0
  - Note that this is no longer the case in Vista, which will explicitly check for stack overflows

The lstrlenW() SEH _except_handler3() structure



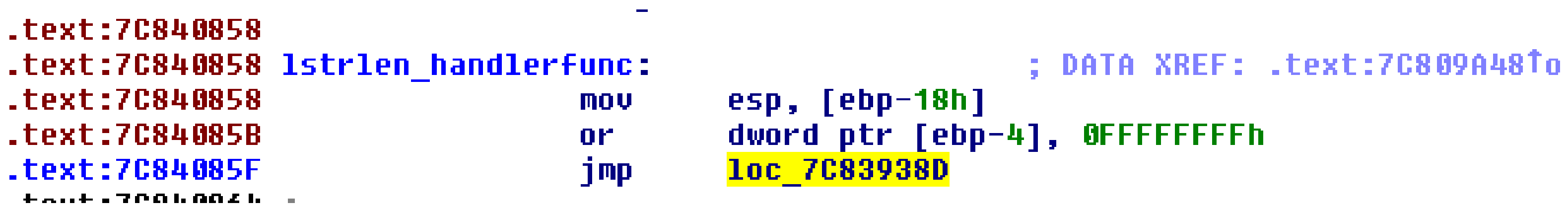
```
.text:7C809A40 dword_7C809A40  dd 0FFFFFFFFh             ; DATA XREF: lstrlenW(x)+2↑o
.text:7C809A44                 dd offset lstrlen_filterfunc
.text:7C809A48                 dd offset lstrlen_handlerfunc
.text:7C809A4C                 dd 90909090h
.text:7C809A50                 db 90h
```

The lstrlenW() Filter Function – catch everything

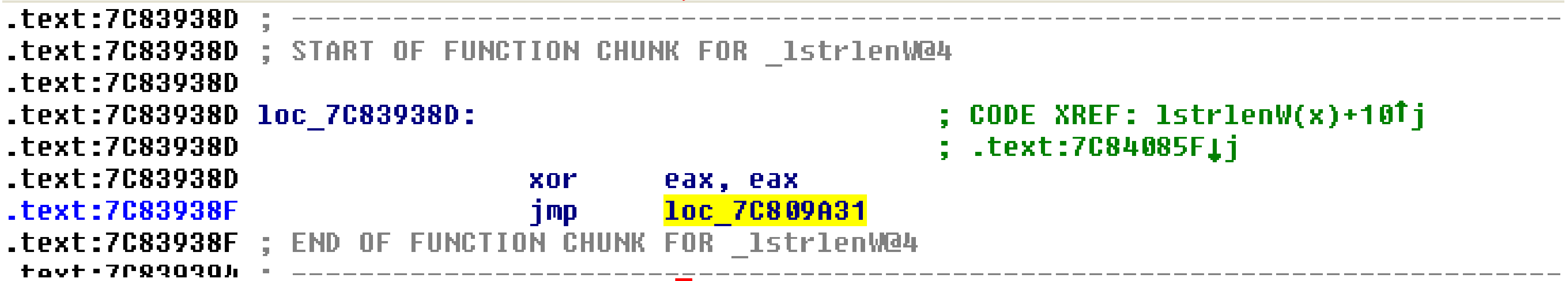


```
.text:7C84084F ; ---------------------------------------------------------------------------
.text:7C84084F
.text:7C84084F lstrlen_filterfunc:                         ; DATA XREF: .text:7C809A44↑o
.text:7C84084F                 xor     eax, eax
.text:7C840851                 inc     eax
.text:7C840852                 retn
.text:7C840852 ; ---------------------------------------------------------------------------
```

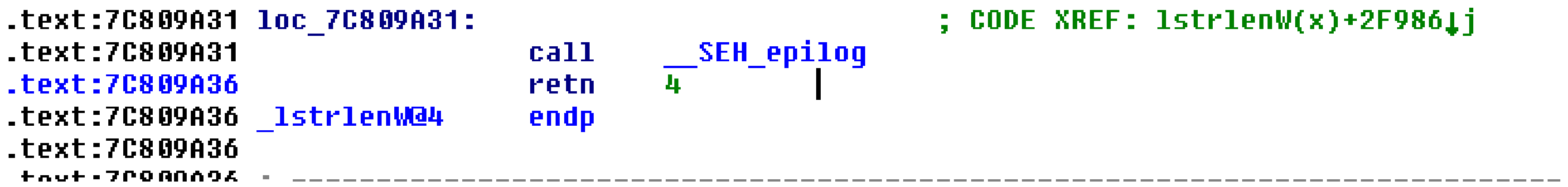
The lstrlenW() Handler Function – return 0



```
.text:7C840858
.text:7C840858 lstrlen_handlerfunc:                                ; DATA XREF: .text:7C809A48↑o
.text:7C840858                 mov     esp, [ebp-18h]
.text:7C84085B                 or      dword ptr [ebp-4], 0FFFFFFFFh
.text:7C84085F                 jmp     loc_7C83938D
```

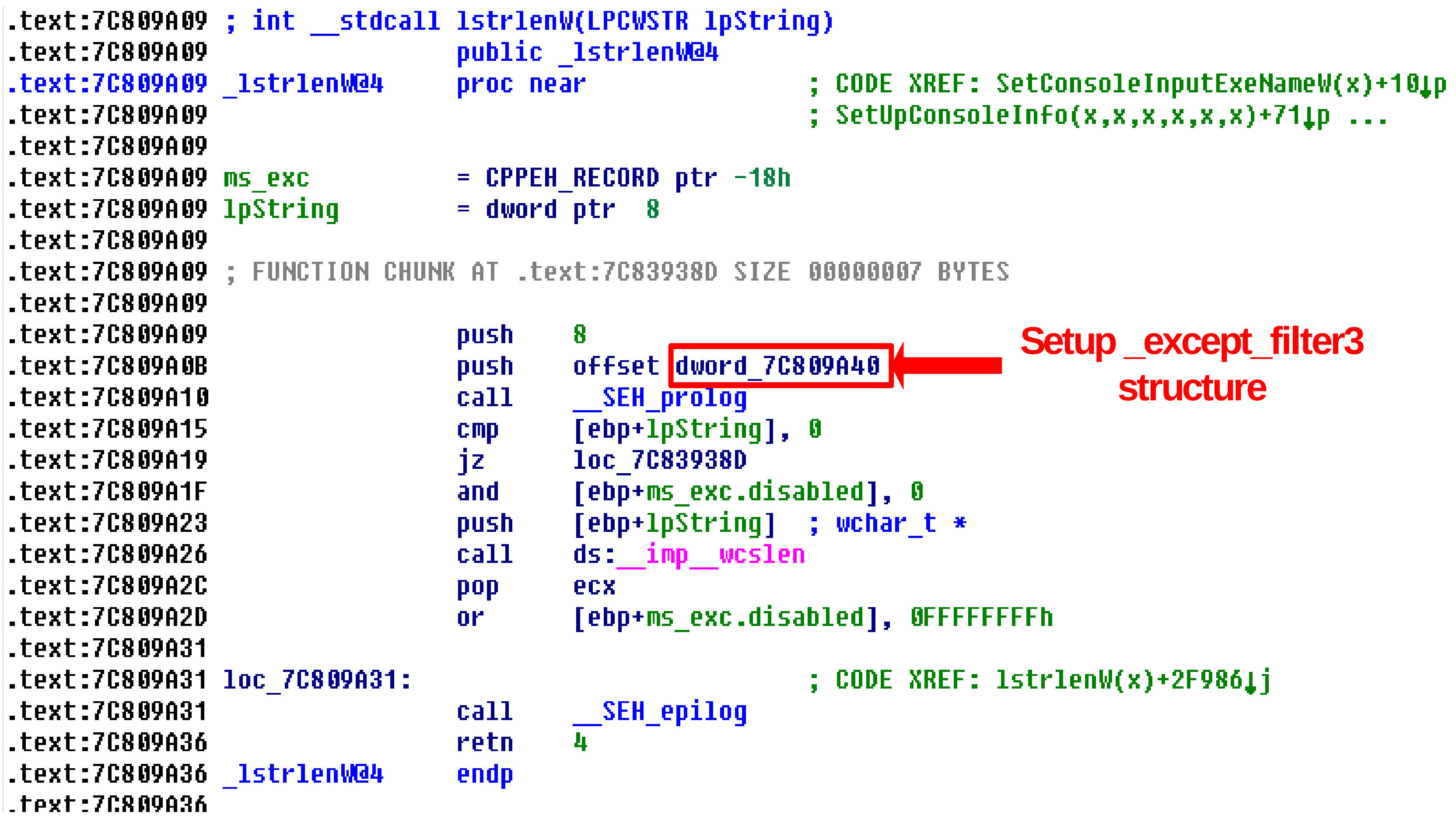


```
.text:7C83938D ; ---------------------------------------------------------------------------
.text:7C83938D ; START OF FUNCTION CHUNK FOR _lstrlenW@4
.text:7C83938D
.text:7C83938D loc_7C83938D:                                       ; CODE XREF: lstrlenW(x)+10↑j
.text:7C83938D                                                     ; .text:7C84085F↓j
.text:7C83938D                 xor     eax, eax
.text:7C83938F                 jmp     loc_7C809A31
.text:7C83938F ; END OF FUNCTION CHUNK FOR _lstrlenW@4
```



```
.text:7C809A31 loc_7C809A31:                                       ; CODE XREF: lstrlenW(x)+2F986↓j
.text:7C809A31                 call    __SEH_epilog
.text:7C809A36                 retn    4
.text:7C809A36 _lstrlenW@4     endp
.text:7C809A36
```

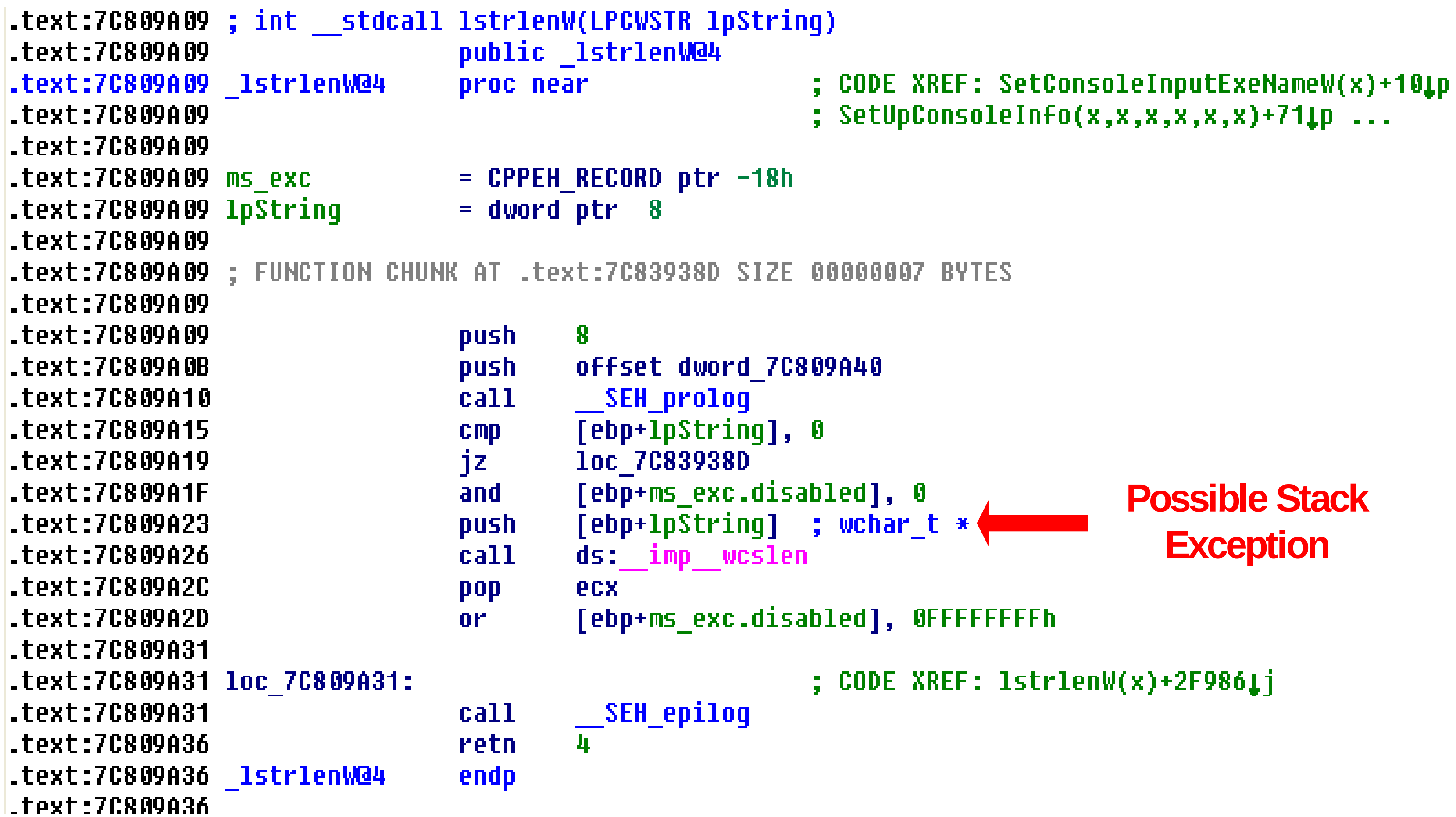
Exploiting Stack Overflow Exceptions – lstrlenW()



```
.text:7C809A09 ; int __stdcall lstrlenW(LPCWSTR lpString)
.text:7C809A09                 public _lstrlenW@4
.text:7C809A09 _lstrlenW@4     proc near               ; CODE XREF: SetConsoleInputExeNameW(x)+10↓p
.text:7C809A09                                         ; SetUpConsoleInfo(x,x,x,x,x,x)+71↓p ...
.text:7C809A09
.text:7C809A09 ms_exc          = CPPEH_RECORD ptr -18h
.text:7C809A09 lpString        = dword ptr  8
.text:7C809A09
.text:7C809A09 ; FUNCTION CHUNK AT .text:7C83938D SIZE 00000007 BYTES
.text:7C809A09
.text:7C809A09                 push    8
.text:7C809A0B                 push    offset dword_7C809A40
.text:7C809A10                 call    __SEH_prolog
.text:7C809A15                 cmp     [ebp+lpString], 0
.text:7C809A19                 jz      loc_7C83938D
.text:7C809A1F                 and     [ebp+ms_exc.disabled], 0
.text:7C809A23                 push    [ebp+lpString]  ; wchar_t *
.text:7C809A26                 call    ds:__imp__wcslen
.text:7C809A2C                 pop     ecx
.text:7C809A2D                 or      [ebp+ms_exc.disabled], 0FFFFFFFFh
.text:7C809A31
.text:7C809A31 loc_7C809A31:                           ; CODE XREF: lstrlenW(x)+2F986↓j
.text:7C809A31                 call    __SEH_epilog
.text:7C809A36                 retn    4
.text:7C809A36 _lstrlenW@4     endp
.text:7C809A36
```

Setup _except_filter3 structure

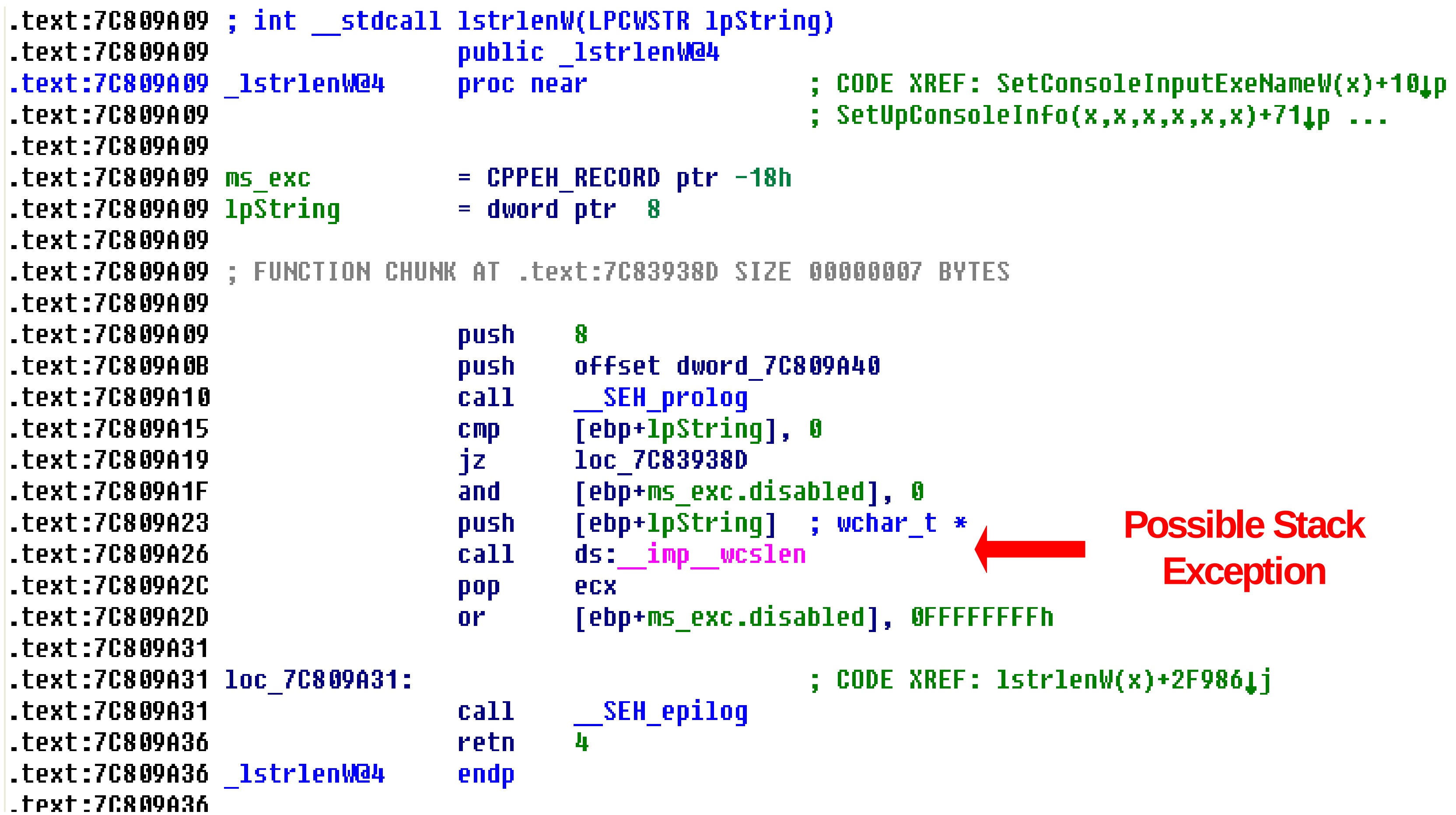
Exploiting Stack Overflow Exceptions – lstrlenW()



```
.text:7C809A09 ; int __stdcall lstrlenW(LPCWSTR lpString)
.text:7C809A09                 public _lstrlenW@4
.text:7C809A09 _lstrlenW@4     proc near               ; CODE XREF: SetConsoleInputExeNameW(x)+10↓p
.text:7C809A09                                         ; SetUpConsoleInfo(x,x,x,x,x,x)+71↓p ...
.text:7C809A09
.text:7C809A09 ms_exc          = CPPEH_RECORD ptr -18h
.text:7C809A09 lpString        = dword ptr  8
.text:7C809A09
.text:7C809A09 ; FUNCTION CHUNK AT .text:7C83938D SIZE 00000007 BYTES
.text:7C809A09
.text:7C809A09                 push    8
.text:7C809A0B                 push    offset dword_7C809A40
.text:7C809A10                 call    __SEH_prolog
.text:7C809A15                 cmp     [ebp+lpString], 0
.text:7C809A19                 jz      loc_7C83938D
.text:7C809A1F                 and     [ebp+ms_exc.disabled], 0
.text:7C809A23                 push    [ebp+lpString]  ; wchar_t *
.text:7C809A26                 call    ds:__imp__wcslen
.text:7C809A2C                 pop     ecx
.text:7C809A2D                 or      [ebp+ms_exc.disabled], 0FFFFFFFFh
.text:7C809A31
.text:7C809A31 loc_7C809A31:                           ; CODE XREF: lstrlenW(x)+2F986↓j
.text:7C809A31                 call    __SEH_epilog
.text:7C809A36                 retn    4
.text:7C809A36 _lstrlenW@4     endp
.text:7C809A36
```

Possible Stack Exception

Exploiting Stack Overflow Exceptions – lstrlenW()



```
.text:7C809A09 ; int __stdcall lstrlenW(LPCWSTR lpString)
.text:7C809A09                 public _lstrlenW@4
.text:7C809A09 _lstrlenW@4     proc near               ; CODE XREF: SetConsoleInputExeNameW(x)+10↓p
.text:7C809A09                                         ; SetUpConsoleInfo(x,x,x,x,x,x)+71↓p ...
.text:7C809A09
.text:7C809A09 ms_exc          = CPPEH_RECORD ptr -18h
.text:7C809A09 lpString        = dword ptr  8
.text:7C809A09
.text:7C809A09 ; FUNCTION CHUNK AT .text:7C83938D SIZE 00000007 BYTES
.text:7C809A09
.text:7C809A09                 push    8
.text:7C809A0B                 push    offset dword_7C809A40
.text:7C809A10                 call    __SEH_prolog
.text:7C809A15                 cmp     [ebp+lpString], 0
.text:7C809A19                 jz      loc_7C83938D
.text:7C809A1F                 and     [ebp+ms_exc.disabled], 0
.text:7C809A23                 push    [ebp+lpString]  ; wchar_t *
.text:7C809A26                 call    ds:__imp__wcslen
.text:7C809A2C                 pop     ecx
.text:7C809A2D                 or      [ebp+ms_exc.disabled], 0FFFFFFFFh
.text:7C809A31
.text:7C809A31 loc_7C809A31:                           ; CODE XREF: lstrlenW(x)+2F986↓j
.text:7C809A31                 call    __SEH_epilog
.text:7C809A36                 retn    4
.text:7C809A36 _lstrlenW@4     endp
.text:7C809A36
```

Possible Stack Exception

Exploiting Stack Overflow Exceptions – API Examples

- **Generating a convenient exception here can lead to problems**
  - Returning 0 for a valid string can help to evade maximum length checks
  - Similarly, if the result is used in an allocation calculation, it can allocate too few bytes, most likely resulting in a buffer overflow
  - Also might lead to incorrectly passed size parameters to functions such as MultiByteToWideChar()
  - This is often the case, since older ATL implementations have W2A() and A2W() macros to convert between CHAR and WCHAR by allocating a buffer on the stack

Exploiting Stack Overflow Exceptions – API Examples

```
DWORD VulnerableFunction1(WCHAR *lpszSource)
{
        WCHAR lpszDestination[MAX_PATH];

        if(lstrlen(lpszSource) >= MAX_PATH)
                return 0;

        lstrcpy(lpszDestination, lpszSource);

        ... more code ...
}

DWORD VulnerableFunction2(WCHAR *lpszSource)
{
        WCHAR *lpszDestination;
        DWORD dwLength;

        dwLength = lstrlen(lpszSource) + 1;

        lpszDestination = (WCHAR *)LocalAlloc(LPTR, dwLength * sizeof(WCHAR));

        if(lpszDestination == NULL)
                return 0;

        lstrcpy(lpszDestination, lpszSource);

        ... more code …
}
```

Exploiting Stack Overflow Exceptions – API Examples

```
DWORD VulnerableFunction1(WCHAR *lpszSource)
{
        WCHAR lpszDestination[MAX_PATH];

        if(lstrlen(lpszSource) >= MAX_PATH)
                return 0;

        lstrcpy(lpszDestination, lpszSource);

        ... more code ...
}

DWORD VulnerableFunction2(WCHAR *lpszSource)
{
        WCHAR *lpszDestination;
        DWORD dwLength;

        dwLength = lstrlen(lpszSource) + 1;

        lpszDestination = (WCHAR *)LocalAlloc(LPTR, dwLength * sizeof(WCHAR));

        if(lpszDestination == NULL)
                return 0;

        lstrcpy(lpszDestination, lpszSource);

        ... more code …
}
```

Exploiting Stack Overflow Exceptions – Other API Calls

- **Other Standard API functions can be similarly afflicted**
  - lstrcpyW() / lstrcatW() use stack space, can result in not doing anything to destination buffers, resulting in uninitialized variable usage
  - In some case, the RtlAllocateHeap() and RtlReallocateHeap() functions can catch exceptions, allowing for arbitrarily small allocations to fail
  - I/O Routines like ReadFile() can be made to fail at an arbitrary point in some cases
  - In some cases, CreateFile() can fail (as well as other functions that use RtlDosPathNameToNtPathName_Ustr(), such as FindFirstFileExW(), CreatePipeName(), etc)
- **Other exceptions may also possibly be generated**
  - Only really likely when there's a bug in the program (divide by zero error, memory corruption, NULL-pointer dereference)
  - Usually, these don't buy you as much, since they can't occur as randomly as stack exhaustion
  - Still, maybe exploitable instances exist (NULL pointer dereferences in particularly)

C++ Exception Handling

- **C++ EH is built on top of SEH**
  - C++ Exceptions have a special exception code reserved (0x0E6D7363) which is used whenever any sort of C++ exception is thrown
  - try { } / catch { } / finally { } are implemented using raw SEH with some specialized functions for figuring out what needs to be done when an exception is thrown
  - _CxxThrowException( ) is used to generate a C++ exception. It builds a parameter list and uses the underlying RaiseException() function to utilize the SEH exception raising mechanism
  - _CxxFrameHandler() is generally used for auto-destruction of stack objects and deciding if any code blocks handle the thrown exception.
  - It utilizes a scope table of sorts to map exceptions to code blocks and so forth
- **Since SEH is used to implement C++ EH, there is some overlap between how EH and SEH behave**
  - C++ exceptions can be caught by native SEH handling / filtering routines, and vice versa
  - This is basically designed not to happen, but the 2 different types of exceptions can have side effects on each other
  - Auto-destruction of stack objects due to system exceptions for example
  - Highly dependant on compiler

Example 1 – Windows SDK basic_string Deallocator

```
void __CLR_OR_THIS_CALL _Tidy(bool _Built = false, size_type _Newsize = 0)
{       // initialize buffer, deallocating any storage
        if (!_Built)
                ;
        else if (_BUF_SIZE <= _Myres)
        {       // copy any leftovers to small buffer and deallocate
                _Elem *_Ptr = _Bx._Ptr;
                if (0 < _Newsize)
                        _Traits_helper::copy_s<_Traits>(_Bx._Buf, _BUF_SIZE, _Ptr, _Newsize);
                _Mybase::_Alval.deallocate(_Ptr, _Myres + 1);
        }

        _Myres = _BUF_SIZE - 1;
        _Eos(_Newsize);
}

union _Bxty
{       // storage for small buffer or pointer to larger one
        _Elem _Buf[_BUF_SIZE];
        _Elem *_Ptr;
} _Bx;
```

Example 1 – Windows SDK basic_string Deallocator

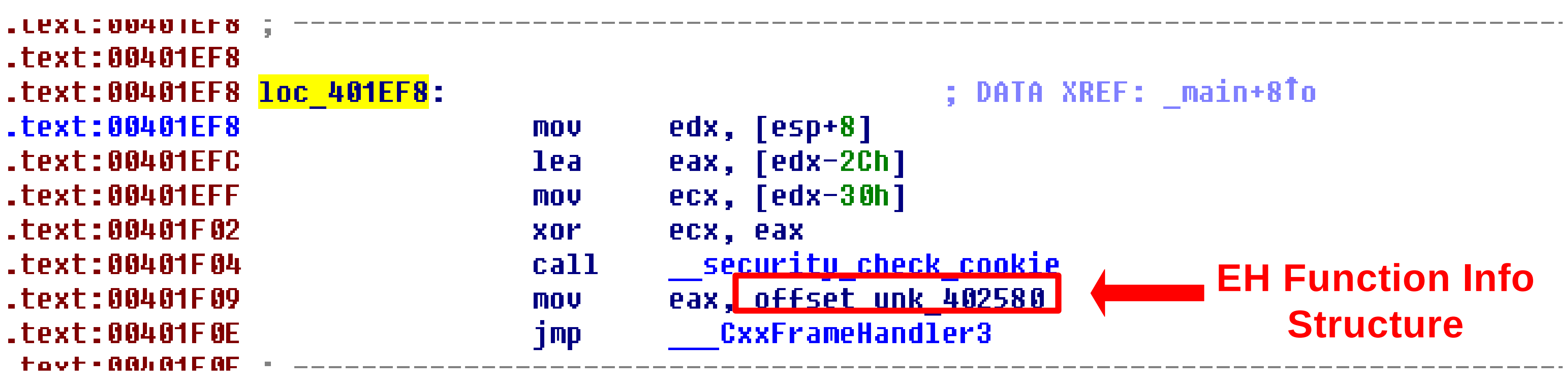
```
void __CLR_OR_THIS_CALL _Tidy(bool _Built = false, size_type _Newsize = 0)
{       // initialize buffer, deallocating any storage
        if (!_Built)
                ;
        else if (_BUF_SIZE <= _Myres)
        {       // copy any leftovers to small buffer and deallocate
                _Elem *_Ptr = _Bx._Ptr;
                if (0 < _Newsize)
                        _Traits_helper::copy_s<_Traits>(_Bx._Buf, _BUF_SIZE, _Ptr, _Newsize);
                _Mybase::_Alval.deallocate(_Ptr, _Myres + 1);
        }

        _Myres = _BUF_SIZE - 1;
        _Eos(_Newsize);
}

union _Bxty
{       // storage for small buffer or pointer to larger one
        _Elem _Buf[_BUF_SIZE];
        _Elem *_Ptr;
} _Bx;
```

C++ Exception Handling

- **C++ Exceptions and System exceptions**
  - Triggered system exceptions can have extra implications in C++-based code
  - Remember that for stack objects, exceptions result in auto-destruction during stack unwinding
  - What if a system exception is generated during the course of a member function?
  - Even if the member function guarantees keeping the object consistent when expected errors occur, unexpected exceptions can result in the object becoming inconsistent
  - Consider the stack exceptions we looked at earlier – they can happen at any point in time pretty much
- **Compiler dictates interaction between system and C++ exceptions**
  - Microsoft Compilers before VS2005 will cause the C++ catch ( ) handler to catch system exceptions as well
  - Also system exceptions will result in stack objects being auto-destructed if the exception isn't caught
  - This is due to how __CxxFrameHandler() packaged with MSVCR80.DLL works (ditto for MSVCRT.DLL on Vista)
  - If the compiler EH Function Info signature is VS2005 (0x19930522) and the low bit of the flags is set (offset 0x20), then it does nothing
  - So, if the compiler is VS2005, system exceptions are ignored by C++ catch( ) blocks unless that flags field is set – ie. program is compiled with /EHa ('a' stands for 'asynchronous exceptions, as opposed to /EHs for synchronous exception handling)

C++ Exception Handling

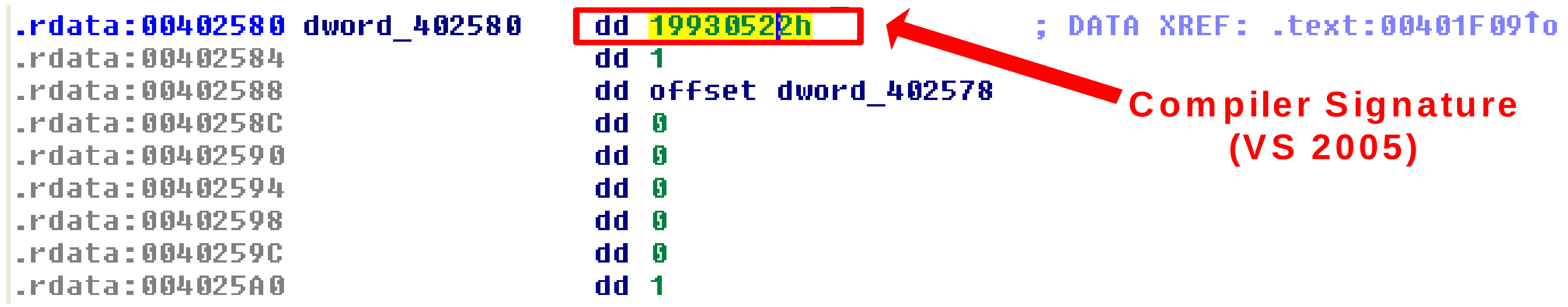
- **Which Compiler/flags were used for a given program?**



```
.text:00401EF8 ; ---------------------------------------------------------------------------
.text:00401EF8
.text:00401EF8 loc_401EF8:                                            ; DATA XREF: _main+8↑o
.text:00401EF8                 mov     edx, [esp+8]
.text:00401EFC                 lea     eax, [edx-2Ch]
.text:00401EFF                 mov     ecx, [edx-30h]
.text:00401F02                 xor     ecx, eax
.text:00401F04                 call    __security_check_cookie
.text:00401F09                 mov     eax, offset unk_402580
.text:00401F0E                 jmp     ___CxxFrameHandler3
.text:00401F0E ; ---------------------------------------------------------------------------
```

C++ Exception Handling

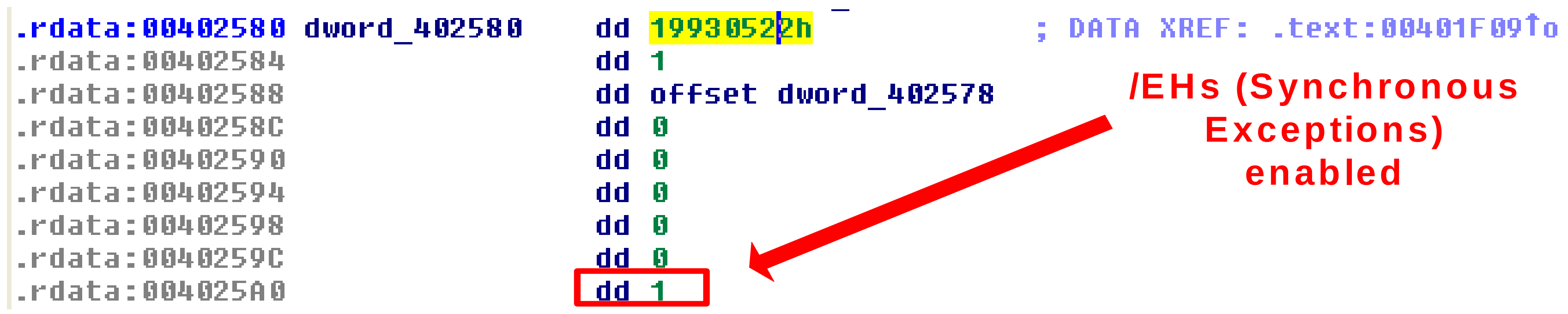
- **Compiler Signature is at the top of the EH Function Info Structure**



```
.rdata:00402580 dword_402580    dd 19930522h              ; DATA XREF: .text:00401F09↑o
.rdata:00402584                 dd 1
.rdata:00402588                 dd offset dword_402578
.rdata:0040258C                 dd 0
.rdata:00402590                 dd 0
.rdata:00402594                 dd 0
.rdata:00402598                 dd 0
.rdata:0040259C                 dd 0
.rdata:004025A0                 dd 1
```

C++ Exception Handling

- **Flag at offset 0x20 indicates if Synchronous EH is enabled**

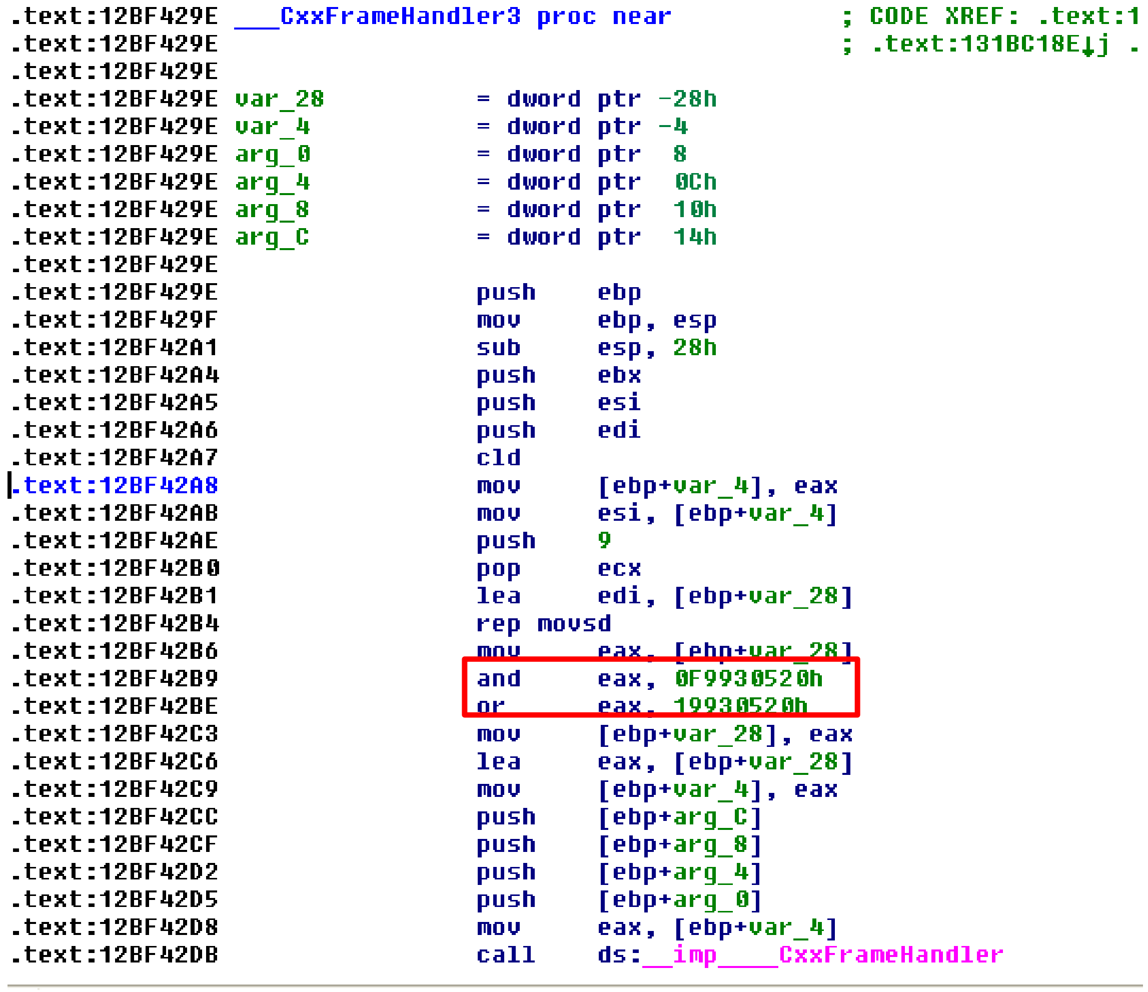


```
.rdata:00402580 dword_402580    dd 19930522h            ; DATA XREF: .text:00401F09↑o
.rdata:00402584                 dd 1
.rdata:00402588                 dd offset dword_402578
.rdata:0040258C                 dd 0
.rdata:00402590                 dd 0
.rdata:00402594                 dd 0
.rdata:00402598                 dd 0
.rdata:0040259C                 dd 0
.rdata:004025A0                 dd 1
```

C++ Exception Handling

- **VS Runtime Library _CxxFrameHandler() implementations and MSVCRT.DLL**
  - All compilers except VS2005 can generate binaries optionally that link to MVVCRT.DLL instead of the appropriate runtime library (example: MSVCR70.DLL)
  - MSVCRT.DLL has a _CxxFrameHandler() implementation that specifically checks for VS6 compiler signature
  - Compilers have to fake it
  - Functionality for _CxxFrameHandler() for each runtime except VS2005 is the same, so it's ok to use this alternate runtime
  - Usually only system binaries use MSVCRT, and they use it almost exclusively
- **Interesting Case: WMP 11 on XP SP2**
  - WMP appears to be compiled with VS2005 with synchronous EH enabled, but is linked against MSVCRT.DLL
  - How does this work?
  - WMP implements a wrapper to _CxxFrameHandler() which copies the EH Function information structure to the stack and switches the compiler signature to VS6's
  - Implications? Using the old MSVCRT _CxxFrameHandler() function means that the synchronous EH functionality is ignored, so this will do asynchronous EH!

WMP Exception Handler



```
.text:12BF429E ___CxxFrameHandler3 proc near          ; CODE XREF: .text:1
.text:12BF429E                                        ; .text:131BC18E↓j .
.text:12BF429E
.text:12BF429E var_28          = dword ptr -28h
.text:12BF429E var_4           = dword ptr -4
.text:12BF429E arg_0           = dword ptr  8
.text:12BF429E arg_4           = dword ptr  0Ch
.text:12BF429E arg_8           = dword ptr  10h
.text:12BF429E arg_C           = dword ptr  14h
.text:12BF429E
.text:12BF429E                 push    ebp
.text:12BF429F                 mov     ebp, esp
.text:12BF42A1                 sub     esp, 28h
.text:12BF42A4                 push    ebx
.text:12BF42A5                 push    esi
.text:12BF42A6                 push    edi
.text:12BF42A7                 cld
.text:12BF42A8                 mov     [ebp+var_4], eax
.text:12BF42AB                 mov     esi, [ebp+var_4]
.text:12BF42AE                 push    9
.text:12BF42B0                 pop     ecx
.text:12BF42B1                 lea     edi, [ebp+var_28]
.text:12BF42B4                 rep movsd
.text:12BF42B6                 mov     eax, [ebp+var_28]
.text:12BF42B9                 and     eax, 0F9930520h
.text:12BF42BE                 or      eax, 19930520h
.text:12BF42C3                 mov     [ebp+var_28], eax
.text:12BF42C6                 lea     eax, [ebp+var_28]
.text:12BF42C9                 mov     [ebp+var_4], eax
.text:12BF42CC                 push    [ebp+arg_C]
.text:12BF42CF                 push    [ebp+arg_8]
.text:12BF42D2                 push    [ebp+arg_4]
.text:12BF42D5                 push    [ebp+arg_0]
.text:12BF42D8                 mov     eax, [ebp+var_4]
.text:12BF42DB                 call    ds:__imp____CxxFrameHandler
```

C++ Exception Handling

- **C++ Exceptions are generally easier to generate**
  - Exceptions used instead of error return codes
  - Erroneous input, memory allocation failures, etc
  - They are expected to happen from time to time and are generally handled better, although not always
  - The most interesting are probably memory allocation errors, since memory allocations happen so frequently, and it's easy to forget about
- **Throwing exceptions != returning an error**
  - Generally, exceptions are just out of band error codes
  - They are harder to ignore, developers need to specifically decide to handle it and ignore it
  - Not handling it generally results in program (controlled) crash
  - Not always harmless – global and stack based objects will be destructed

C++ Exception Handling

```
class Module
{
private:
      Credentials *m_credentials;

public:
      Module()
            :m_credentials(NULL);

      ~Module()
      {
            if(m_credentials)
                  delete(m_credentials);

            m_credentials = NULL;
      }

      bool Authenticate(unsigned int authType, unsigned char *authenticationBlob)
      {
            if(m_credentials)
                  delete m_credentials;

            m_credentials = new Credentials(authType, authenticationBlob);

            ... more code ...
      }
};
```

C++ Exception Handling

```
class Module
{
private:
      Credentials *m_credentials;

public:
      Module()
            :m_credentials(NULL);

      ~Module()
      {
            if(m_credentials)
                  delete(m_credentials);

            m_credentials = NULL;
      }

      bool Authenticate(unsigned int authType, unsigned char *authenticationBlob)
      {
            if(m_credentials)
                  delete m_credentials;

            m_credentials = new Credentials(authType, authenticationBlob);

            ... more code ...
      }
};
```

Auditing For Stack Overflow Problems

- **Find Codepaths with all necessary ingredients present**
  - Unrestricted stack usage – alloca() is preferable because it allows far more precision
  - Otherwise, recursive function calls might be of some use (depending on the “exception gap” you are trying to hit)
  - From that point on, you must find a function with useful exception handling functionality – API ones we talked about, or custom exception handling
  - Functionality you’re looking for might involve causing an allocation failure, a miscalculation (like with lstrlen), or more likely, an uninitialized variable usage scenario
- **But wait, it’s not quite that easy…**
  - You only get one shot at a stack overflow exception
  - The function with the interesting EH must be called at the peak of stack usage
  - Previously called functions with bigger stack footprints affect whether or not there is actually an exploitable problem
  - You need to ensure that the function you call has a bigger stack frame or is called at a point deeper into the stack than previous function calls
  - This can be difficult and time consuming to work out

Auditing For C++ Exception Problems

- **Identify Compilation Environment**
  - Compilation environment affects whether system exceptions will be caught by C++ catch() blocks
  - Identify compilation environment using methods discussed earlier in the speech (mainly checking EH function info structure and which version of _CxxFrameHandler() is linked)
- **When performing traditional audits on member functions, note which exceptions are caught**
  - Making a special note of catch-all exception handlers
  - Check what happens when exceptions are caught – the most likely problems to occur will be uninitialized variable usage or possibly double de-allocation style problems
  - Another implication can be a function not performing the duty it’s supposed to – for a string assignment where exceptions are caught, is the result that the string is never assigned?
- **Record all the functions that can throw exceptions**
  - Functions that can generate exceptions may occur deeply within a call chain
  - Knowing which ones can throw exceptions will be useful for finding subtle EH-related bugs
  - Remember allocation exceptions!
  - You should familiarize yourself with standard STL functions that can throw them as well when they are used

Auditing For C++ Exception Problems

Example Exception Audit Log

Function	throws	catches	passes	Notes
Thing::Thing()	invalid_dimensions		bad_alloc from new[]	Seems ok
Thing::resize()	invalid_dimensions resize_failed	bad_alloc		Possible inconsistent state if invalid_dimensions is thrown – check callers
Thing::render()	render_failed	*catch-all* - (rethrows render_failed)		If we get a system exception in the call to render(), might be improperly handled by the catch(…)

Auditing For Exception Problems - Automation

- **Automation of some of the key elements of these vulnerabilities is possible**
  - Discovering vulnerabilities of this nature particularly in binaries can be difficult
  - Non-linear code paths, exception handling code is often ignored
  - Manually checking scope tables to see what regions of code are protected by what filter/handler can be a little tiresome and easy to forget about
  - For stack exceptions, finding dynamic allocations and recursion is really time consuming
  - Checking stack footprints around where the allocation happens too is hard and easy to get wrong
  - Lots of this stuff is relatively easy to automate
- **Summary of desirable features**
  - Easy identification of code blocks guarded with exception handlers/filters
  - Identification of any dynamic stack allocations
  - Identification of catch-all exception handlers
  - Stack footprint enumeration
  - Might want functionality for when looking at one binary, or across a whole directory of binaries

Auditing For Exception Problems - Automation

- **Identifying exception handlers**
  - Easy to do – functions with SEH will typically begin with _SEH_prolog or similar
  - Single argument is a pointer to a structure which contains the scope table (and possibly cookie offsets in the case of SEH4)
  - Code in function can be parsed for modifications to the tryLevel stack variable, and used to determine which exception filters guard which parts of the code
  - Nesting can be identified using the scope table
- **Dynamic stack allocations**
  - Some compilers subtract directly from ESP, but VS uses a function to do it safely - __chkstk/_alloca_probe/_alloca_probe_8/_alloca_probe_16
  - Find this function and cross-reference all instances to it
  - The amount of space it will allocate is passed in EAX
  - Check if EAX contains a variable value, if so it is interesting
  - Doesn't really require in-depth dataflow analysis, it always gets populated shortly before the call
  - Usually with an immediate value if its not a variable allocation (and also occasionally with an immediate value that's been pushed to the stack and retrieved with a pop instruction)

Auditing For Exception Problems - Automation

- **Catch-all Exception Handlers**
  - These might require some level of dataflow analysis, depending on how in depth you want to go
  - Usually the filter expression is a small isolated piece of code
  - Typical catch-all exception handlers zero out eax (with xor eax, eax) and then increment it (inc eax)
- **Stack footprints**
  - These can sometimes be more difficult to enumerate
  - Functions other than those in the relevant function need to be evaluated in addition to code portions leading up to the interesting function call
  - Different code paths leading to the interesting function call might use different amount of stack, and as such might be vulnerable depending on which code path is taken through the function
  - Dataflow analysis is not needed per se, but understanding stack changes in all scenarios is necessary

Exception Handling and Exploitation

- **Exception handlers being present allows for control-data modification in memory corruption conditions**
  - EH records on the stack enable exploitation in many case
  - Very well-known technique – overwriting exception handler structures with pointers to arbitrary addresses
  - Microsoft addressed this by adding “SafeSEH” functionality
  - Litchfield discussed this in a previous blackhat speech in 2003 (https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf)
  - Methods included: returning to memory not located within any module, returning to code in modules without a safe SE handler table, abusing the registered exception handlers
  - Microsoft has cleaned a lot of these vectors up, but some ideas are still valid
  - Abusing registered exception handlers is possible, if a module exports any dangerous functionality

Exception Handling and Exploitation

- **Exploiting _except_handler3**
  - Many modules register _except_handler3(), as it was the SEH function for code built with compilers earlier than VS2005
  - Litchfield mentioned abusing this function, but additional functionality has since been put in place to validate the stack frame
  - __ValidateEH3RN() responsible for validation
  - Ryan Smith (member of X-Force) devised some ways of gaining execution bypassing the protections

Exception Handling and Exploitation

- **Exception handling might also facilitate in the exploitation of vulnerabilities**
  - When the exception handler itself is not insecure, but the operations it performs if an exception is triggered might aid an attacker
  - Doing things like catching memory access violations and recovering can be particularly beneficial for an attacker exploiting a memory corruption style vulnerability
  - This was the case in the recent ANI vulnerability as detailed by Alex Sotirov (http://www.determina.com/security.research/vulnerabilities/ani-header.html)
  - Scenarios like this allow for multiple attempts at exploiting a bug successfully
  - Improves reliability, undermines some protections such as ASLR. cookies
  - When exploiting a bug, you need to determine what exception handlers are present, and carefully consider the code being executed given that the process is potentially compromised (ie, has a smashed stack/heap/etc)

Thank you!

markdowd@au1.ibm.com
neelm@us.ibm.com
jrmcdona@us.ibm.com
IBM ISS X-Force R&D

IBM Internet Security Systems
Ahead of the threat.™