Brandon Bray (branbray@microsoft.com)

Safe Exceptions and Compiler Security Checks

Brandon Bray
Program Manager
Microsoft Visual C++ Compiler

April 2, 2003

Agenda

- Under the Hood of Security Checks
  - Technical Background
  - How Cookies Work
  - Common Misconceptions
  - Safe Exceptions
  - Walking Through Exploits
  - Reacting to an Attack

Buffer overruns are expensive

Buffer overruns are studied

- Buffer overruns were once documented!
- Learn simple exploits in an hour
- Every security book covers the subject
- Education on buffer overruns is naïve
- Most time is spent on preventing buffer overruns
  - Testing and code review is a sieve; what if some slip through?

Brandon Bray (branbray@microsoft.com)

Visual C++'s Goal

Transform a buffer overrun from an extremely bad security danger to an unacceptable nuisance

- Make a program withstand an attack even in the presence of a buffer overrun
- We are far from this goal

What is a buffer overrun?

- The ability to arbitrarily corrupt memory
- Overflows lead to arbitrary code
- Underflows lead to denial of service
- Problem is usually isolated to C and C++

```
int x = 42;
char zip[6];
strcpy(zip, userinput);
printf("x = %i\n", x);
```

2A	00	00	00
00	00	00	00
00	00	00	00

Brandon Bray (branbray@microsoft.com)

Anatomy of the stack

Previous function's stack frame
Function arguments
Return address
Frame pointer
EH frame
Local variables and locally declared buffers
Callee save registers
Garbage

- x86 stacks grow downward
- A buffer overrun on the stack can always rewrite the:
  - Return address
  - Frame pointer
  - EH frame

Types of exploits

- Stack smashing
- Register hijacking
- Local pointer subterfuge
- V-Table hijacking
- C++ EH clobbering
- SEH clobbering
- Multistage attacks
- Parameter pointer subterfuge

Previous function's stack frame
Function arguments
Return address
Frame pointer
EH frame
Local variables and locally declared buffers
Callee save registers
Garbage

Exploit difficulty

- Stack smashing is always possible
- Not every exploit is always possible
  - Attacking a code address is easiest
  - Attacking a data address is harder
  - Exploiting scalar data (not a base for memory indirection) is the hardest

Unsafe APIs

- Many historical APIs of the C standard library are bad
  - `strcpy` has no knowledge of the array size
  - `strncpy` cannot validate the array size
  - Many more unsafe APIs exist
- Static analysis tools are helpful
- Impossible to guarantee a safe API

Run-time checks overview

- In VC6 it is /GZ, in VC7 it is /RTC1
- There are three kinds of run-time checks
  - /RTCs does stack checks
  - /RTCu finds unitialized variables
  - /RTCc catches conversions that truncate information
- /RTC1 is an alias for /RTCsu
- Compiler injects code into the program
- Not intended for production code

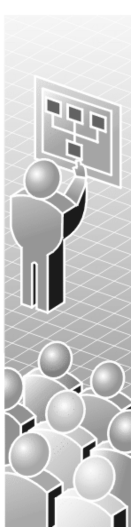
Run-time checks details

- What does /RTCs do?
  - Fills the whole stack with 0xCCCCCCCC
  - Pads all multibyte or address taken variables with four to seven bytes
  - Finds mismatched calling conventions
- What does /RTCu do?
  - Finds positive cases of C4701 warning

A retail solution

- Return address hijacking is always available and the easiest to exploit
- Idea: put a speed bump between the locally declared buffer and the return address
- All of this is done with the /GS switch
  - Windows builds with /GS
  - Visual Studio builds with /GS
  - .NET Developer Platform builds with /GS

Demonstration: Security Checks



In this demonstration, you will learn how to:

- Recompile code with /GS
- React to buffer overrun

Stack layout in VC++ .NET

Function prolog:

```
sub    esp,24h
mov    eax,dword ptr
  [___security_cookie (408040h)]
xor    eax,dword ptr [esp+24h]
mov    dword ptr [esp+20h],eax
```

Function epilog:

```
mov    ecx,dword ptr [esp+20h]
xor    ecx,dword ptr [esp+24h]
add    esp,24h
jmp    __security_check_cookie
  4010B2h)
```

Previous function's stack frame
Function arguments
Return address
Frame pointer
Cookie
EH frame
Local variables and locally declared buffers
Callee save registers
Garbage

Stack layout in VC++ 2003

Function prolog:

```
sub    esp,24h
mov    eax,dword ptr
  [___security_cookie (408040h)]
mov    dword ptr [esp+20h],eax
```

Function epilog:

```
mov    ecx,dword ptr [esp+20h]
add    esp,24h
jmp    __security_check_cookie
  4010B2h)
```

Previous function's stack frame
Function arguments
Return address
Frame pointer
Cookie
EH frame
Locally declared buffers
Local variables
Callee save registers
Garbage

- Requires optimized build

When do we need a cookie?

- Not every function is vulnerable
- Cookie is put on the stack only when a local object contains a buffer where:
  - Buffer has more than four bytes of storage
  - Buffer elements are one or two bytes each

What is this cookie?

- Generated by the function `__security_init_cookie`
- Original stored in the variable `__security_cookie`
- Cookie is random (at least 20 bits)
- Cookie is per image and generated at load time
- Cookie is the size of a pointer

Common problems

- Calling `_CRT_INIT` while security checked functions are live
  - Only temporary workarounds exist

```
DllEntryPoint(...) {
   char buf[10]; // triggers security check
   ...
   _CRT_INIT();
   ...
}
```

- Predictable cookie when no CRT init
  - Solved with Windows Server 2003 and VC7.1

Performance impact

- Expect less than a 2% degradation
- Most application did not notice anything
- With both VC7 and VC7.1 the improvements in optimization far outweigh the cost of security checks
- Each security check is nine instructions

> **"The perf hit hasn't shown up for us. There was no test hit associated with the change. The only cost we've had associated with this is getting ourselves to build with /GS.**
> **– IIS6 Developer**

Security checks philosophy

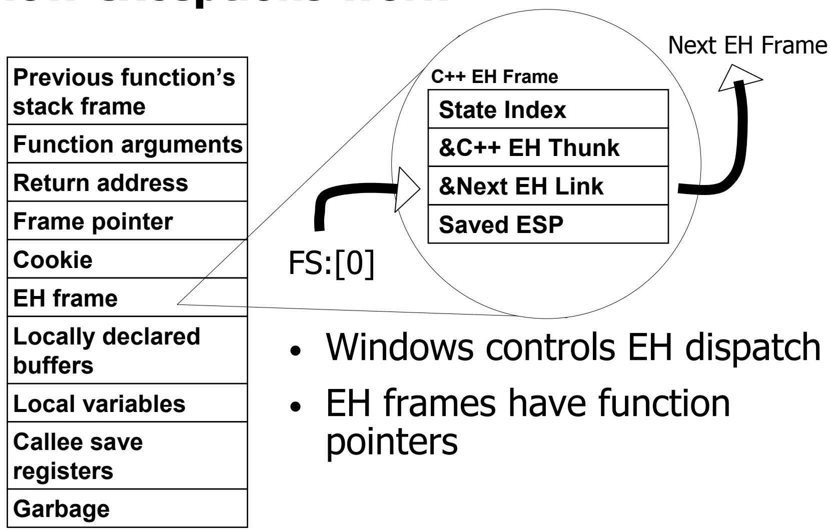
- It is NOT okay to knowingly have buffer overruns in your code!
- Faulty code is the program's fault, not the fault of security checks architecture
- /GS is an insurance policy
- /GS attempts to protect you from some of the unprotected buffers you missed
- Both VC7 and VC7.1 have limited abilities

Armchair critics

- Just use good functions
- My code is perfect
- It is a trade for denial of service
- STL solves the problem
- The real problem is not solved
- More avenues of attack exist
- Image size explodes
- Bad code is tolerated and encouraged

Brandon Bray (branbray@microsoft.com)

How exceptions work



- Windows controls EH dispatch
- EH frames have function pointers

All exception attacks

- Sequence of events
  - Cause a buffer overrun and overwrite the EH frame
  - The address of the handler points to an arbitrary address
  - Cause an exception to occur
  - The operating system follows the FS:[0] link to find the address of the handler
  - Windows passes control to the handler

How Code Red worked

- All the attack code was on the stack
  - Windows XP will not dispatch to the stack
- The exception handler was actually an instruction, CALL EBX, in msvcrt.dll
  - EBX stored an address on the stack
  - Windows XP clears out some registers
  - This would have stopped Code Red
  - Not all registers can be cleansed

Safe exceptions overview

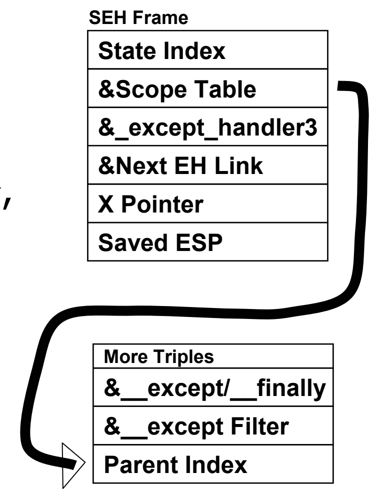
- Visual C++ 2003 creates a table with a list of all the handlers in the compiland
- Before dispatching to any handler, Windows checks against the list
- If the address is not in the data list, the process terminates
- Check to see if an image is safe:

```
D:\>dumpbin /loadconfig /headers t.exe
   ...
   00406CC0 Safe Exception Handler Table
          5 Safe Exception Handler Count
```

Brandon Bray (branbray@microsoft.com)

SEH specific attacks

- SEH handler is always `_except_handler3`
- A scope table stores the specific code for `__finally`, `__except`
- Attack will try to spoof the scope table



SEH validation in VC2003

- SEH handler enforces the following:
  - Scope table is in read only memory
  - Checks the shape of the scope table
  - Parent indexes must be sound
- If the validation fails, the handler tells Windows to terminate the process

Stack smashing (VC .NET)

```
#define BUFLEN 4

void vulnerable(void) {
  wchar_t buf[BUFLEN];
  int val;

  val = MultiByteToWideChar(
    CP_ACP, 0, "1234567",
    -1, buf, sizeof(buf));
  printf("%d\n", val);
}
```

Attack Code
Hijacked EIP
Garbage **with invalid cookie**
Garbage

Stack smashing (VC2003)

```
#define BUFLEN 4

void vulnerable(void) {
  wchar_t buf[BUFLEN];
  int val;

  val = MultiByteToWideChar(
    CP_ACP, 0, "1234567",
    -1, buf, sizeof(buf));
  printf("%d\n", val);
}
```

Attack Code
Hijacked EIP
Garbage **with invalid cookie**
val
Garbage

V-Table hijacking (VC .NET)

```
class Vulnerable {
public:
  int value;
  Vulnerable() {value=0;}
  virtual ~Vulnerable()
    {value=-1;}
};

void vulnerable(char* str) {
  Vulnerable vuln;
  char buf[20];
  strcpy(buf, str);
}
```

Attack Code
Hijacked V-Table
&Hijacked V-Table
Garbage
Garbage

V-Table hijacking (VC2003)

```
class Vulnerable {
public:
  int value;
  Vulnerable() {value=0;}
  virtual ~Vulnerable()
    {value=-1;}
};

void vulnerable(char* str) {
  Vulnerable vuln;
  char buf[20];
  strcpy(buf, str);
}
```

Attack Code
Garbage with invalid cookie
&Vulnerable V-Table
vuln
Garbage

Pointer subterfuge (VC .NET)

```
void vulnerable(
  char* buf, int cb)
{
  char name[8];
  void (*func)() = foo;

  memcpy(name, buf, cb);
  (func)();
}
```

Attack Code
&Attack Code
Garbage
Garbage

Pointer subterfuge (VC2003)

```
void vulnerable(
  char* buf, int cb)
{
  char name[8];
  void (*func)() = foo;

  memcpy(name, buf, cb);
  (func)();
}
```

Attack Code
Garbage **with invalid cookie**
&foo
Garbage

EH clobbering (VC .NET)

```
int vulnerable(char* str) {
  char buf[8];
  char* pch = str;
  strcpy(buf, str);
  return *pch == '\0';
}

int main(
  int argc, char* argv[]) {
  __try {
    vulnerable(argv[1]);
  } __except(2) { return 1; }
  return 0;
}
```

Attack Code
Hijacked EH frame
Garbage
0xBFFFFFFF
Garbage
Garbage

EH clobbering (VC2003)

```
int vulnerable(char* str) {
  char buf[8];
  char* pch = str;
  strcpy(buf, str);
  return *pch == '\0';
}

int main(
  int argc, char* argv[]) {
  __try {
    vulnerable(argv[1]);
  } __except(2) { return 1; }
  return 0;
}
```

Attack Code
Hijacked EH frame
Garbage with invalid cookie
&pch
Garbage

Brandon Bray (branbray@microsoft.com)

The main handler

```
void __cdecl __security_error_handler(
  int code, void *data)
{
  if (user_handler != NULL) {
    __try {
      user_handler(code, data);
    } __except (EXCEPTION_EXECUTE_HANDLER) {}
  } else {
    // ...prepare outmsg...
    __crtMessageBoxA(
      outmsg,
      "Microsoft Visual C++ Runtime Library",
      MB_OK|MB_ICONHAND|
        MB_SETFOREGROUND|MB_TASKMODAL);
  }
  _exit(3);
}
```

Installing a user handler

- Defined in stdlib.h

```
void __cdecl report_failure(
  int code, void * unused)
{
  if (code == _SECERR_BUFFER_OVERRUN)
    printf("Buffer overrun detected!\n");
}

void main()
{
  _set_security_error_handler(
    report_failure);
  ...more code...
}
```

What to do in a user handler

- Do not raise exceptions
- Do not call DebugBreak
- Do not longjmp
- Hook up to error reporting
- Just print your own message
- Do not trust any data in the process

Rewriting the main handler

- DO NOT replace the function `__security_error_handler`
  - Many smart people have tried and failed
  - This is tricky and it has to be right
- Use `_set_security_error_handler`
- Do not avoid terminating the program
  - Nothing can be trusted
  - The only safe thing to do is terminate the entire process

Exploitations still available

- Parameter pointer subterfuge
- Two stage attacks
- Local objects with buffers
- Heap attacks

Hardware support

- Windows tracks execute, writable permissions for each page of memory
- x86 does not enforce execution in PTE
- IA64 and AMD64 do enforce these
  - Stack is not executable
  - Some security checks on 64-bit needed
  - Visual C++ does not yet have /GS for 64-bit
- x86 may enforce permissions someday

Brandon Bray (branbray@microsoft.com)

Questions