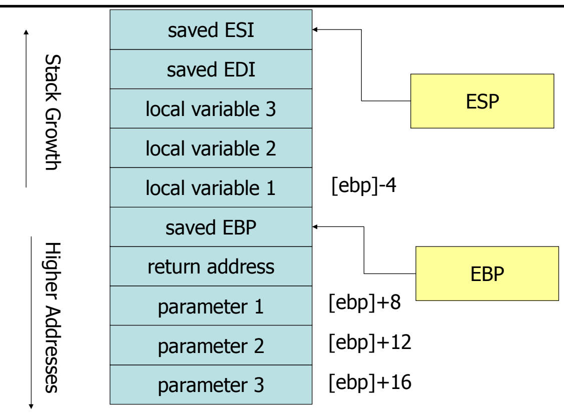
CS216: Program and Data Representation
University of Virginia Computer Science
Spring 2006 David Evans

Lecture 22: Unconventional Calling

http://www.cs.virginia.edu/cs216

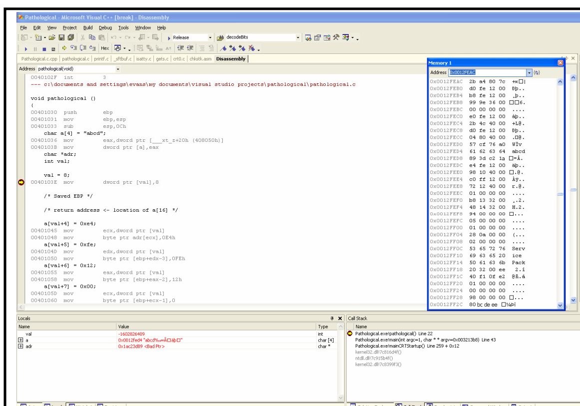
Menu

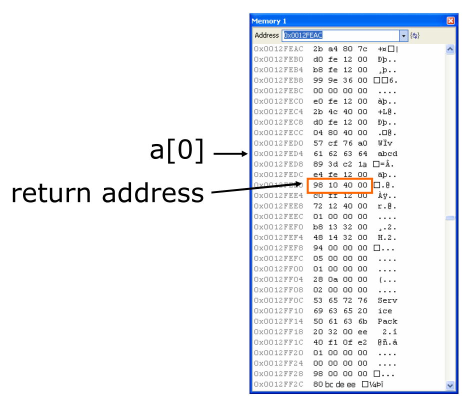
- Stack Smashing Attacks and Defenses
- ISR De-Randomizing MicroVM



Pathological C Program

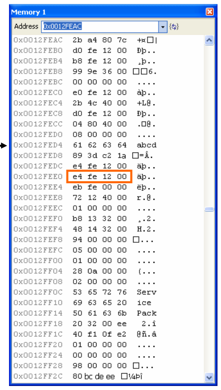
```
void pathological ()
{
        char a[4] = "abcd";
        char *adr;
        int val;
        val = 8;
        /* Saved EBP */
        /* return address <- location of a[16] */
        a[val+4] = 0xe4;
        a[val+5] = 0xfe;
        a[val+6] = 0x12;
        a[val+7] = 0x00;
        /* a[16-17] <- JMP -2 */
        a[val+8] = 0xeb;
        a[val+9] = 0xfe;
        a[val+10] = 0x00;
        a[val+11] = 0x00;
}

int main (int argc, char **argv)
{
        pathological ();
        printf ("Hello\n");
        return EXIT_SUCCESS;
}
```





```
void pathological ()
{
        char a[4] = "abcd";
        char *adr;
        int val;
        val = 8;
        a[val+4] = 0xe4;
        a[val+5] = 0xfe;
        a[val+6] = 0x12;
        a[val+7] = 0x00;
        /* a[16-17] <- JMP -2 */
        a[val+8] = 0xeb;
        a[val+9] = 0xfe;
        a[val+10] = 0x00;
        a[val+11] = 0x00;
}
```

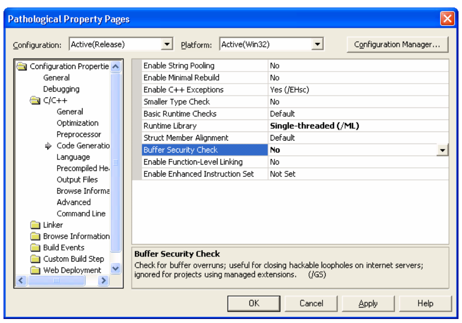
a[0] →




```
0040107D  mov        edx,dword ptr [val]
00401080  mov        byte ptr [ebp+edx+3],0
}
00401085  mov        esp,ebp
00401087  pop        ebp
00401088  ret
```

```
0012FEE4  jmp        0012FEE4
```


/GS



/GS Option

The compiler injects checks in functions with local string buffers or, on x86, functions with exception handling. A string buffer is defined as an array whose element size is one or two bytes, and where the size of the whole array is at least five bytes, or, any buffer allocated with _alloca.

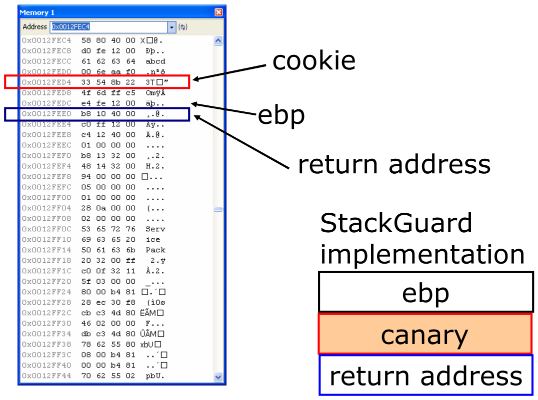
http://msdn2.microsoft.com/en-US/library/8dbf701c(VS.80).aspx

With /GS

```
void pathological ()
{
  char a[5] = "abcd";
```


Security Cookies

```
void pathological ()
{
00401030  push   ebp
00401031  mov    ebp,esp
00401033  sub    esp,14h
00401036  mov    eax,dword ptr [___security_cookie (408060h)]
0040103B  mov    dword ptr [ebp-8],eax
...
0040109B  mov    eax,dword ptr [val]
0040109E  mov    byte ptr [ebp+eax-5],0
}
004010A3  mov    ecx,dword ptr [ebp-8]
004010A6  call   __security_check_cookie (40111Eh)
004010AB  mov    esp,ebp
004010AD  pop    ebp
004010AE  ret
```



Cookie Checking

```
void __declspec(naked) __fastcall
__security_check_cookie(DWORD_PTR cookie)
{
   /* x86 version written in asm to preserve all regs */
   __asm {
      cmp ecx, __security_cookie
0040111E  cmp        ecx,dword ptr [___security_cookie (408060h)]
      jne failure
00401124  jne        failure (401127h)
      ret
00401126  ret
failure:
      jmp report_failure
00401127  jmp        report_failure (4010EDh)
```


Does it work?

```
void pathological ()
{
        char a[5] = "abcd";
        char *adr;
        int val;
        printf ("Hello\n");
        val = 16;
        /* return address <- location of a[16] */
        a[val+4] = 0xe4;
        a[val+5] = 0xfe;
        a[val+6] = 0x12;
        a[val+7] = 0x00;

        /* a[16-17] = JMP -2 */
        a[val+8] = 0xeb;
        a[val+9] = 0xfe;
        a[val+10] = 0x00;
        a[val+11] = 0x00;
}
```

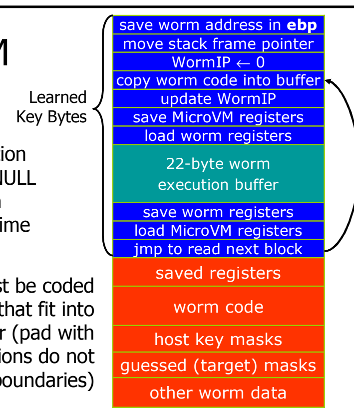

Works in Practice?

- Most attacks can't skip over cookie
  - Must fill up buffer, instead of directly assigning to locations
- Lots and lots of other proposed defenses

An Instruction Set Randomizing MicroVM

MicroVM

76 bytes of code
\+ 22 bytes for execution
\+ __2 bytes to avoid NULL
= 100 bytes is enough
> 99% of the time



Worm code must be coded in blocks that fit into execution buffer (pad with noops so instructions do not cross block boundaries)

MicroVM Code

```
push dword ebp   mov ebp, WORM_ADDRESS + WORM_REG_OFFSET
pop dword [ebp + WORM_DATA_OFFSET]
xor eax, eax         ; WormIP = 0 (load from ebp + eax)
read_more_worm: ; read NUM_BYTES at a time until worm is done
  cld        xor ecx, ecx            mov byte cl, NUM_BYTES
  mov dword esi, WORM_ADDRESS     ; get saved WormIP
  add dword esi, eax              mov edi, begin_worm_exec
  rep movsb                       ; copies next Worm block into execution buffer
  add eax, NUM_BYTES              ; change WormIP
  pushad                          ; save register vals
  mov edi, dword [ebp]            ; restore worm registers
  mov esi, dword [ebp + ESI_OFFSET]      mov ebx, dword [ebp + EBX_OFFSET]
  mov edx, dword [ebp + EDX_OFFSET]      mov ecx, dword [ebp + ECX_OFFSET]
  mov eax, dword [ebp + EAX_OFFSET]
begin_worm_exec:                  ; this is the worm execution buffer
  nop nop nop nop nop nop nop nop nop nop nop nop
  nop nop nop nop nop nop nop nop nop nop nop nop
  mov [ebp], edi     ; save worm registers
  mov [ebp + ESI_OFFSET], esi          mov [ebp + EBX_OFFSET], ebx
  mov [ebp + EDX_OFFSET], edx          mov [ebp + ECX_OFFSET], ecx
  mov [ebp + EAX_OFFSET], eax
  popad              ; restore microVM register vals
  jmp read_more_worm
```

Note: this is nasm x86 assembly code, not masm, so some directives are slightly different

Charge

- PS7 Due Wednesday
  - Lots of interesting things to explore
- Exam 2 will be posted Thursday
- Next class: review (***if*** you send questions)
- “x86 Guru” title(s) may be awarded for especially clever answers to Question 10