CSE 227
Computer Security

Winter 2012

Stefan Savage

Today

- Catching up
- Projects
- Review of software vulnerabilities

Projects

- Some kind of research project in security
- Best in a group of two
  - If you can’t find a partner I’ll be willing to consider single person projects; but I prefer if this is the exception
- Please form groups in the next week (Jan 30th)
  - Send me mail by next Mon identifying who is in your group
- Initial project proposals due Feb 13th
  - One page
  - What you plan to do, Why is it interesting, How you’ll do it, What you’re not sure about (or what resources you need)
- Ultimately 6 pages and short talk (10-15mins)
- Hope: some sufficiently interesting to be real paper

Generally speaking

- Most projects will fall into the category of:
  - **Analysis**: evaluate the security of a system of interest
  - **Attack**: identify some new attack/vulnerability, develop/test it and discuss the possible ramifications, mitigations, etc
  - **Measurement**: measure some aspect of adversarial behavior (real or potential), characterize it, explore its limits, etc
  - **Design/Implementation**: design and/or build a new system that addresses a problem in a new way

Things to thing about...

- Pick good problems
  - Why is this problem interesting or will become interesting?
  - Look at what others are doing:
    - Academic conferences: USENIX Security, ACM CCS, IEEE S&P, NDSS, DIMVA, RAID, LEET, WOOT, PET, eCrime
    - Non-academic conferences: BlackHat, Defcon, HITB, ShmooCon, various blogs
- Pick problems that are achievable
  - What resources would you need to investigate the problem?
- Think about how to evaluate your work

Random ideas

- On class Web page

- This is not a list you must pick from!
- Just examples to give you ideas and make sure you understand how broad the scope is

Resources

- Servers
- Lots of SPAM (1M/day?), lots of related data (DNS data, rendered web pages, etc)
- Lots of malware samples, lots of twitter
- DDoS data, traffic traces, netflow data, university power monitors
- Lots of 802.11 gear (192 radios throughout the building), directional antennas, oscilloscopes and logic probes
- Big chunks of Internet address space
- Fingerprinting supplies,
- Good DSLR, pro-am HDTV camera, telescope
- Lots of low-level stuff in the embedded lab
- Legal clearance for various kinds of purchasing activity

- Ask if you’re serious and you need something

Questions about project?

What's a software vulnerability?

- A bug in a software program that allows an unprivileged user capabilities that should be denied to them

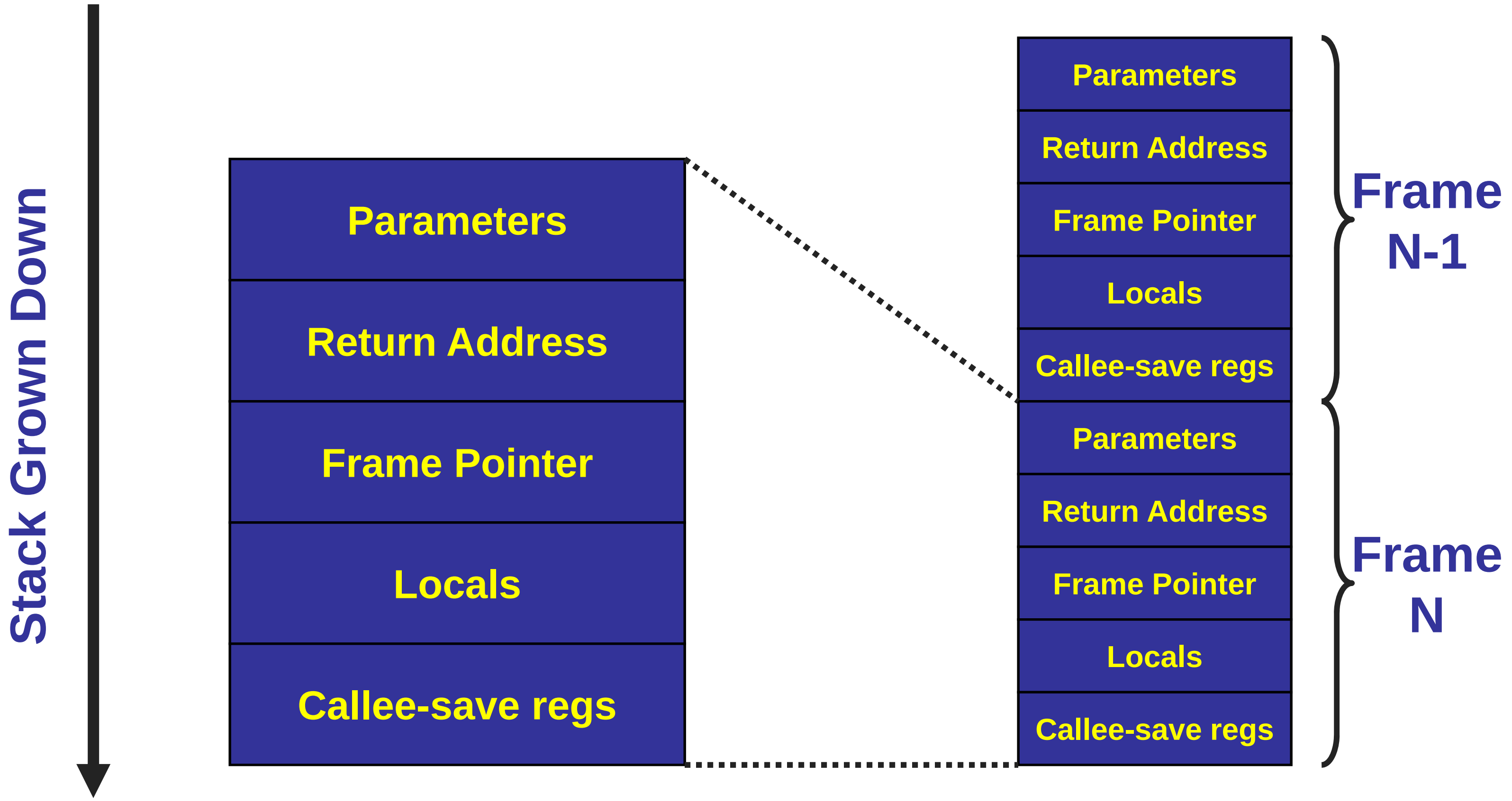
- Worst kind?
  - Control hijacking
    - Divert control flow (in instruction stream)
    - Divert to "payload" that executes code of adversary's choosing

Classic: Stack overflows

- Robert T. Morris worm, 1988
  (note: not control data)

- Cannon
  - AlephOne “Hacking the Stack for Fun and Profit”, Phrack 49, 1996
  - Dildog, “The Tao of Windows Buffer Overruns”, Cult of The Dead Cow cDC-351, 1998
  - Overwrite control data on stack to execute arbitrary instructions from input

Recap: Stack activations for C



Stack Grown Down

Parameters

Return Address

Frame Pointer

Locals

Callee-save regs

Parameters

Return Address

Frame Pointer

Locals

Callee-save regs

Frame N-1

Parameters

Return Address

Frame Pointer

Locals

Callee-save regs

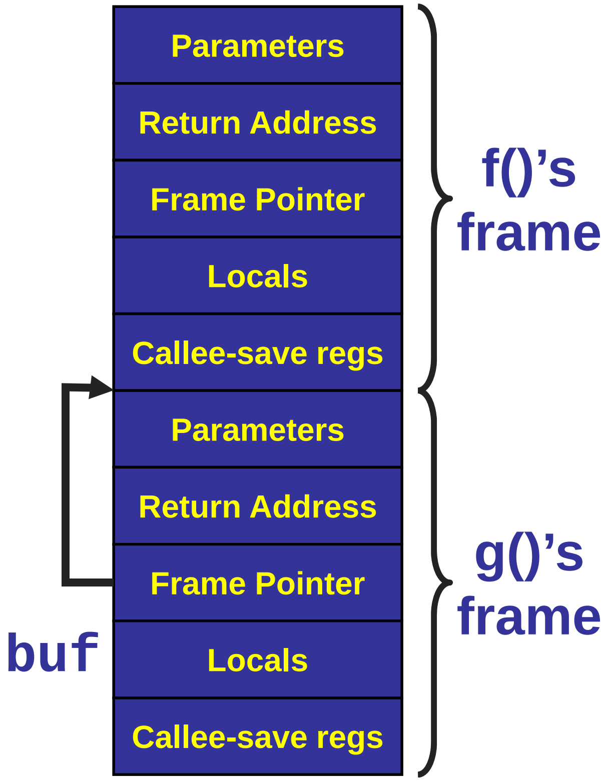
Frame N

Example

```
f() {
  g(parameter);
}

g(char *string) {
  char buf[16];

  strcpy(buf,string);
}
```



What this looks like (Windows x86 cdecl call)

Prolog

```
push ebp    // save old frame pointer
mov ebp,esp // Set current frame pointer
sub esp,10h // reserve 16 bytes for buf
push ebx    //callee saves
push esi
push edi
```

```
…  do stuff
```

Epilog

```
pop edi // restore callee saves
pop esi
pop ebx
mov esp,ebp // unroll stack
pop ebp //restore old frame pointer
ret 3 // pop eip and jmp to it
```

Caveat: no opt, no /GZ, no /GS

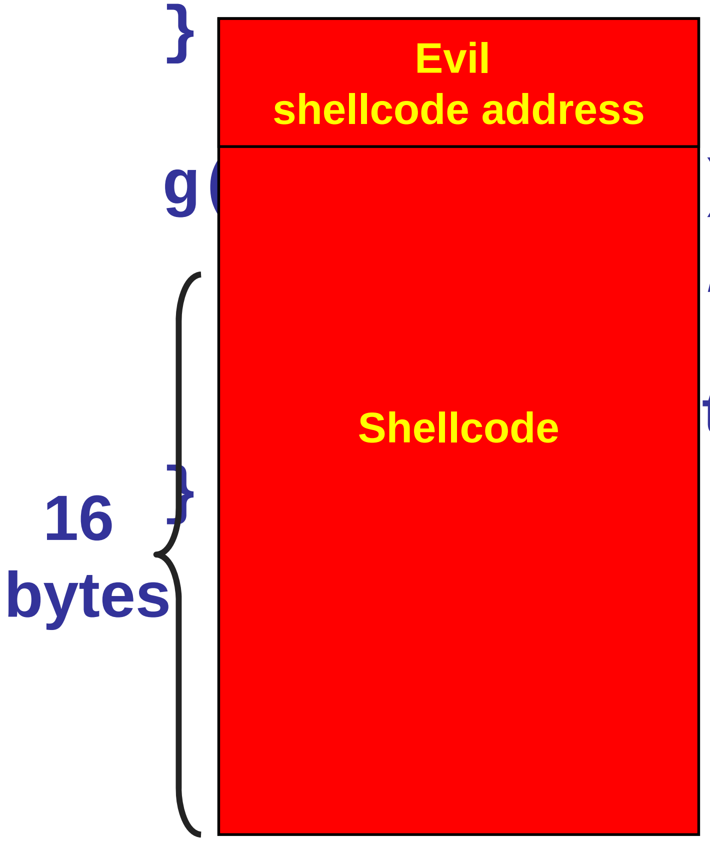
Quintessential stack overflow

- Basic problem is that the library routines look like this:

  ```
  void strcopy(char *src, char *dst) {
    int i = 0;
    while (src[i] != “\0”) {
      dst[i] = src[i];
      i = i + 1;
    }
  }
  ```
- If the memory allocated to dst is smaller than the memory needed to store the contents of src, a buffer overflow occurs
- Particularly problematic with c’s idiom of using local temporary buffers – allows “stack smashing” attack

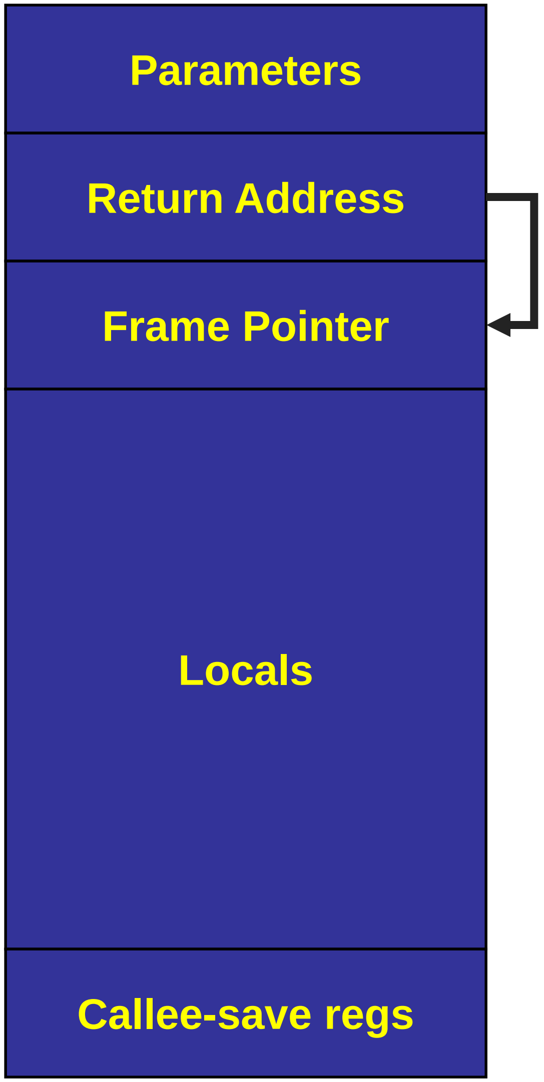
Stack smashing in action

```
f() {
  g(badstring);
}
```



```
g(          ) {
            ;

            tring)
}
```

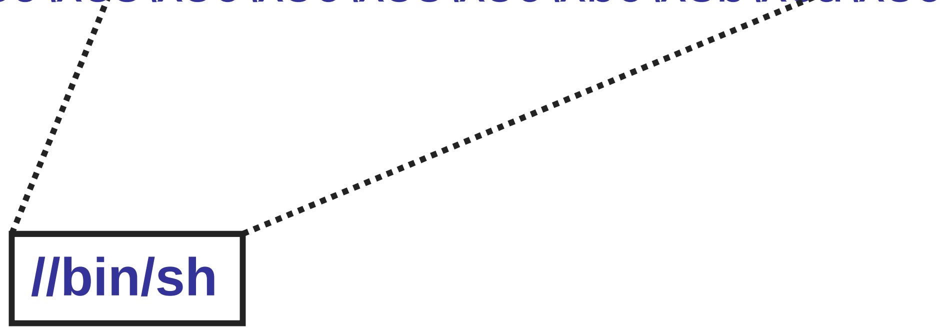


Aside: why is it called shellcode?

```
xor     eax, eax
push    eax
push    0x68732f6e
push    0x69622f2f
mov     ebx, esp
push    eax
push    eax
push    ebx
mov     al, 59
push    eax
int  80h
```

System Call (execve)

```
char shellcode[] =
"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62”
“\x69\x89\xe3\x50\x50\x53\x50\xb0\x3b\xcd\x80";
```



Shellcode courtesy Foster, Osipov, Bhalla and Heinen

Vulnerabilities, threats and hindsight

- Just a bug or exploitable vulnerability?
- Lots of hot air expended on this topic
  - “Yes, you found a bug, but its not exploitable”
  - “This class of bugs is very hard to exploit”
  - “While the DoS threat is significant, this vulnerability can’t be used for code injection”
- Historically these distinctions have changed with experience
  - Case in point: the off-by-one stack overflow
  - Historically, not considered a major control hijacking threat
  - Today, considered easy

Off-by-one example

```
main() {
   f();
}

f() {
   g(input);
}

g(char *input) {
   char buf[16];
   int i;
   for (i=0;i<=16;i++)
      buf[i]=input[i];
}
```

function epilog

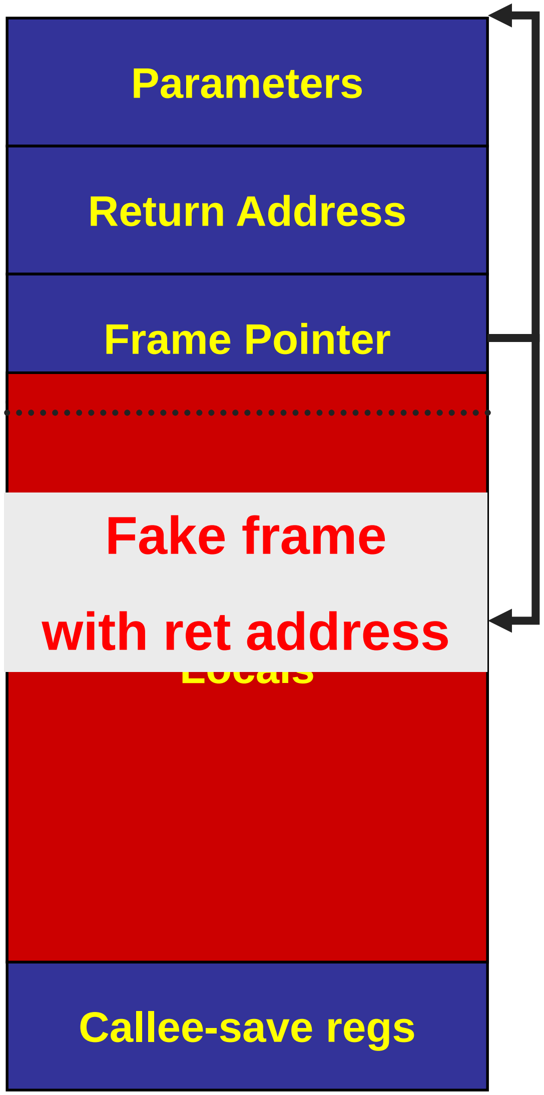
…

mov esp,ebp // unroll stack

…

When **f** returns control hijacked

Can overflow buffer by 1 byte!



Integer overflow: Classic example

What if:
len1 == 0xFFFFFFFE
len2 == 0x000000102

```
void *ConcatBytes(void *buf1, unsigned int len1,
                  char *buf2, unsigned int len2)
{
  void *buf = malloc(len1 + len2);
  if (buf == NULL) return;

  mem    (buf, buf1, len1);
  me     (buf + len1, buf2, len2);
```

0x100 bytes allocated… not enough. Ooops.

Courtesy Jon Pincus

Aside: vulnerability research is “trendy”

- Example
  - Integer overflow reports from National Vulnerability Database
  - Zalewski identifies Integer overflow in OpenSSH in March of 2001
    - One more found 4 months later (tcpdump)
    - 8 in 2002, 27 in 2003, 41 in 2004, 66 in 2005, 96 in 2006, 126 in 2007
- Common pattern
  - Once new “class” of vulnerability is identified, then it gets found everywhere
- XP SP2 impact

Generic heap overflow

- Key idea: heap data structures holds both data and metadata (where allocated chunks are)
- The metadata holds pointers
  - Linked lists typically (allocated chucks vs free list)
- Heap impl writes **through** those pointers

- If you overwrite heap data into pointers you can control both the address and value

Typical problem (simplified)

- Each allocated memory chunk has a header

•prev (ptr)	•next (ptr)
•Data	

- Used to track allocated/free memory
- Removing a block (a) from a list

```
a.prev->next = a.next;
a.next->prev = a.prev;
```

- What if you overwrite data block?
- Write arbitrary data to arbitrary location

Language ambiguity: Bitfields

- C/C++ allow bit-level data types

```
struct {
   unsigned int a:8 (8 bits)
} b;
```

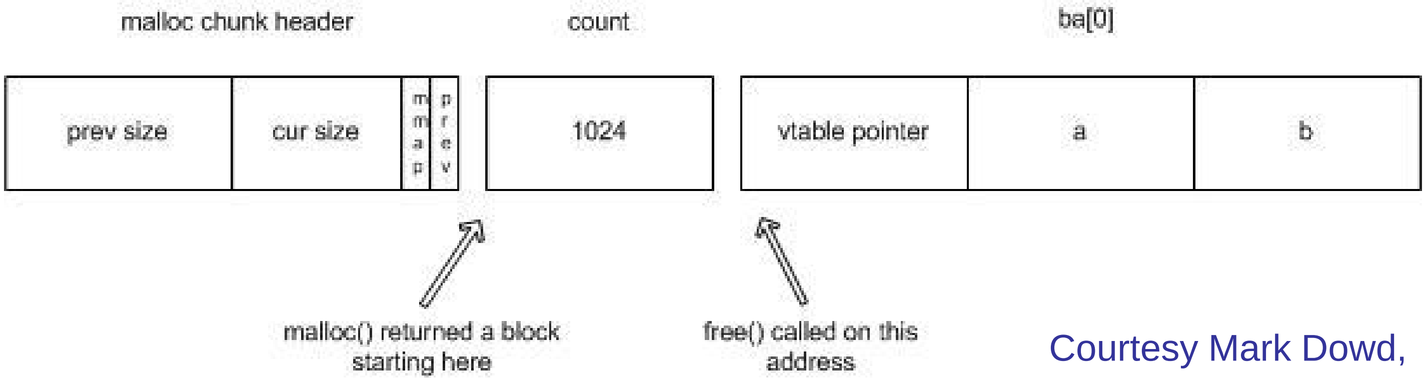
  - Typically used to map onto bit-level file/stream formats
  - Vagueness in the standard leads to problems
- Truncation
  - Not clear how to handle bitfield as an rvalue (c = b.a)
    - » Gcc model: use length of type (i.e., int = 32 bits)
    - » MSVC model: use length of bitfield (i.e. 8 bits)
- Sign conversion
  - What is type of b.a?
  - Not defined by standard, but many implementations implement it as a signed number!
- Bottom line: trivial to get this wrong

Language ambiguity: delete and delete[]

- Arrays of objects allocated/deallocated with new[] and delete[] in C++; not new and delete
- Incorrect code:

```
int main(void) {
    basebob *ba = (basebob *) new bob[1024];
    dostuff(ba);
    delete ba;
 }
```

- Minor issue: only destructor for ba[0] is called
- Bigger problem: different heap representation



Courtesy Mark Dowd, Vulnerabilities II

Attacks and Defenses

- What are the essential elements of control flow hijacking?
- What could you do to defend against or mitigate it?
- What could you do to go around those things?

Kinds of defenses

- Eliminate violation of runtime model
  - » Better languages, code analysis
- Don’t allow bad input
  - » Input validation
- Detect overflow/overwrite of data structures
  - » Stack validation
  - » Run-time bounds checking, pointer validation, etc
  - » Reference monitors
- Don’t allow untrusted code to execute
  - » Hardware protection, code signing
- Minimize invariants for making repeatable exploits
  - » ASLR, code randomization, encrypted pointers
- Minimize impact of untrusted code running

Other kinds of low-level software attacks

- » Return-to-libc
  - Chained function calling
  - Return-oriented programming
- » Don’t know buf’s address
  - Trampolining (don’t know buff’s address)
  - NOP sleds
- » Other kinds of overwrites
  - Function pointer clobbering
  - Data pointer overwrite (4 byte with/that)
  - Vtables, exception handlers
  - Format string
  - Heap overflow, heap spray
  - Type conversions
- » Multi-stage attacks

This is just the surface…

- If you’re into this stuff,
  - Read Kotler’s “Advanced Buffer Overflow Methods” for more shellcode hacks
    - E.g. using program literals as serendipitous instructions; jumping into middle of instructions, etc
  - Read Dowd et al’s “Art of Software Security Assessment” for more nasty C/C++ issues (they also update a blog with new ones)

More stuff you could be reading…

- The important vulnerability research literature is generally **not** from academia

- To keep up to date
  - Dave Aitel (Daily Dave mailing list)
  - H.D. Moore (browserfun.blogspot.com & metasploit)
  - Halvar Flake (ADD/XOR/ROL)
  - Matasano blog (general)
  - Blackhat Briefings talks and some of the other cons

My favorite unintuitive interactions

```
BOOL DoStuff() {
   char pPwd[64];
   size_t cchPwd = sizeof(pPwd) / sizeof(pPwd[0]);
   BOOL fOK = false;
   if (GetPassword(pPwd, &cchPwd))
      fOK = DoSecretStuff(pPwd, cchPwd);
   memset(pPwd, 0, sizeof(pPwd));
   return fOK;
}
```

When DoStuff() returns can you still find the password on the stack?

Yes, compiler optimizes call to memset away…

Courtesy Mike Howard

My favorite unintuitive interaction

```
BOOL DoStuff() {
   char pPwd[64];
   size_t cchPwd = sizeof(pPwd) / sizeof(pPwd[0]);
   BOOL fOK = false;
   if (GetPassword(pPwd, &cchPwd))
      fOK = DoSecretStuff(pPwd, cchPwd);
   memset(pPwd, 0, sizeof(pPwd));
   *(volatile char*)pPwd = *(volatile char *)pPwd;
   return fOK;
}
```

Prevent optimization. Volatile tells compiler ptr can be changed/accessed outside program scope

Courtesy Mike Howard

For next time...

- Look at two kinds of defense papers
  - CFI– low-level control flow isolation
  - Nozzle – system for detecting heap spraying attacks