Part IV: Software

Why Software?

- Why is software as important to security as crypto, access control, protocols?
- Virtually all of information security is implemented in software
- If your software is subject to attack, your security can be broken
  - Regardless of strength of crypto, access control or protocols
- Software is a poor foundation for security

Chapter 11: Software Flaws and Malware

If automobiles had followed the same development cycle as the computer,
a Rolls-Royce would today cost $100, get a million miles per gallon,
and explode once a year, killing everyone inside.
— Robert X. Cringely

My software never has bugs. It just develops random features.
— Anonymous

Bad Software is Ubiquitous

- NASA Mars Lander (cost $165 million)
  - Crashed into Mars due to…
  - …error in converting English and metric units of measure
  - Believe it or not
- Denver airport
  - Baggage handling system --- very buggy software
  - Delayed airport opening by 11 months
  - Cost of delay exceeded $1 million/day
  - What happened to person responsible for this fiasco?
- MV-22 Osprey
  - Advanced military aircraft
  - Faulty software can be fatal

Software Issues

Alice and Bob

- Find bugs and flaws by accident
- Hate bad software…
- …but must learn to live with it
- Must make bad software work

Trudy

- Actively looks for bugs and flaws
- Likes bad software…
- …and tries to make it misbehave
- Attacks systems via bad software

Complexity

- "Complexity is the enemy of security", Paul Kocher, Cryptography Research, Inc.

System	Lines of Code (LOC)
Netscape	17 million
Space Shuttle	10 million
Linux kernel 2.6.0	5 million
Windows XP	40 million
Mac OS X 10.4	86 million
Boeing 777	7 million

- A new car contains more LOC than was required to land the Apollo astronauts on the moon

Lines of Code and Bugs

- Conservative estimate: 5 bugs/10,000 LOC
- **Do the math**
  - Typical computer: 3k exe's of 100k LOC each
  - Conservative estimate: 50 bugs/exe
  - So, about 150k bugs per computer
  - So, 30,000-node network has 4.5 billion bugs
  - Maybe only 10% of bugs security-critical and only 10% of those remotely exploitable
  - Then "only" 45 million critical security flaws!

Software Security Topics

- Program flaws (unintentional)
  - Buffer overflow
  - Incomplete mediation
  - Race conditions
- Malicious software (intentional)
  - Viruses
  - Worms
  - Other breeds of malware

Program Flaws

- An **error** is a programming mistake
  - To err is human
- An error may lead to incorrect state: **fault**
  - A fault is internal to the program
- A fault may lead to a **failure**, where a system departs from its expected behavior
  - A failure is externally observable

error → **fault** → **failure**

Example

```
char array[10];
for(i = 0; i < 10; ++i)
     array[i] = `A`;
array[10] = `B`;
```

- This program has an **error**
- This error might cause a **fault**
  - Incorrect internal state
- If a fault occurs, it might lead to a **failure**
  - Program behaves incorrectly (external)
- We use the term **flaw** for all of the above

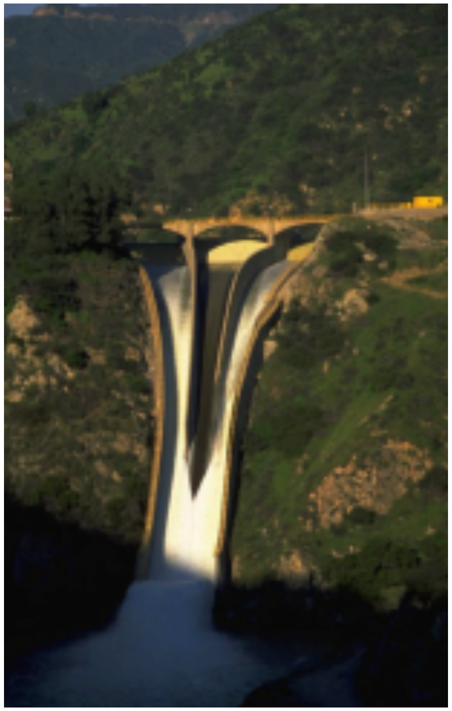
Secure Software

- In software engineering, try to ensure that a program does what is intended
- ***Secure*** software engineering requires that software **does what is intended…**
- **…and nothing more**
- Absolutely secure software is impossible
  - But, absolute security ***anywhere*** is impossible
- How can we manage software risks?

Program Flaws

- Program flaws are **unintentional**
  - But can still create security risks
- We'll consider 3 types of flaws
  - Buffer overflow (smashing the stack)
  - Incomplete mediation
  - Race conditions
- These are the most common problems

Buffer Overflow



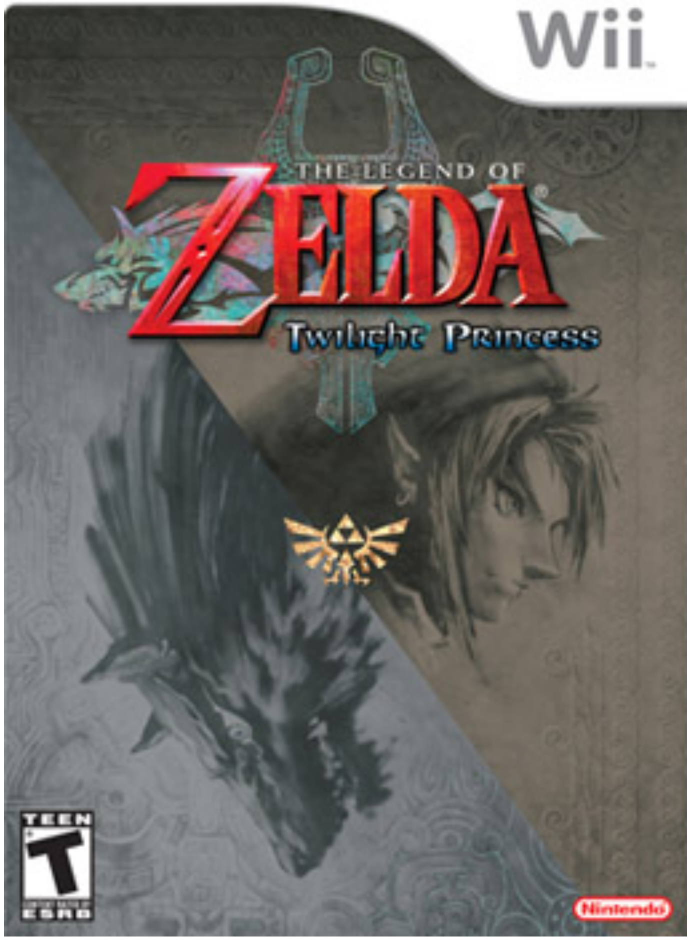
The Twilight Hack

The problem: gamers wanted to create their own games for Nintendo's Wii...

...but Nintendo did not want them to do that.

Horsing Around

- In "The Legend of Zelda: Twilight Princess", the hero gets a horse.
- You can rename the horse, but there is a buffer overflow flaw.



WiiBrew

- With the right name, the WII reboots and reads from an SD card.
- This exploit allowed users to run WiiBrew and play custom Wii games.

Possible Attack Scenario

- Users enter data into a Web form
- Web form is sent to server
- Server writes data to array called buffer, without checking length of input data
- Data "overflows" buffer
  - Such overflow might enable an attack
  - If so, attack could be carried out by anyone with Internet access

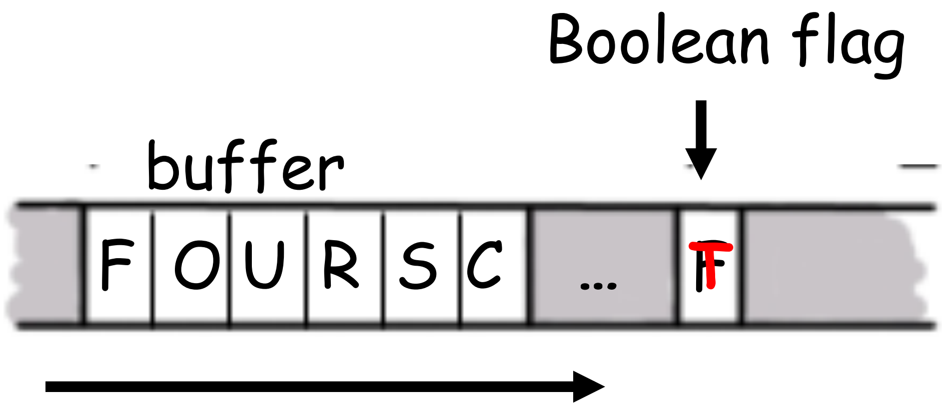
Buffer Overflow

```
int main(){
    int buffer[10];
    buffer[20] = 37;}
```

- **Q:** What happens when code is executed?
- **A:** Depending on what resides in memory at location "buffer[20]"
  - Might overwrite **user** data or code
  - Might overwrite **system** data or code
  - Or program could work just fine

Simple Buffer Overflow

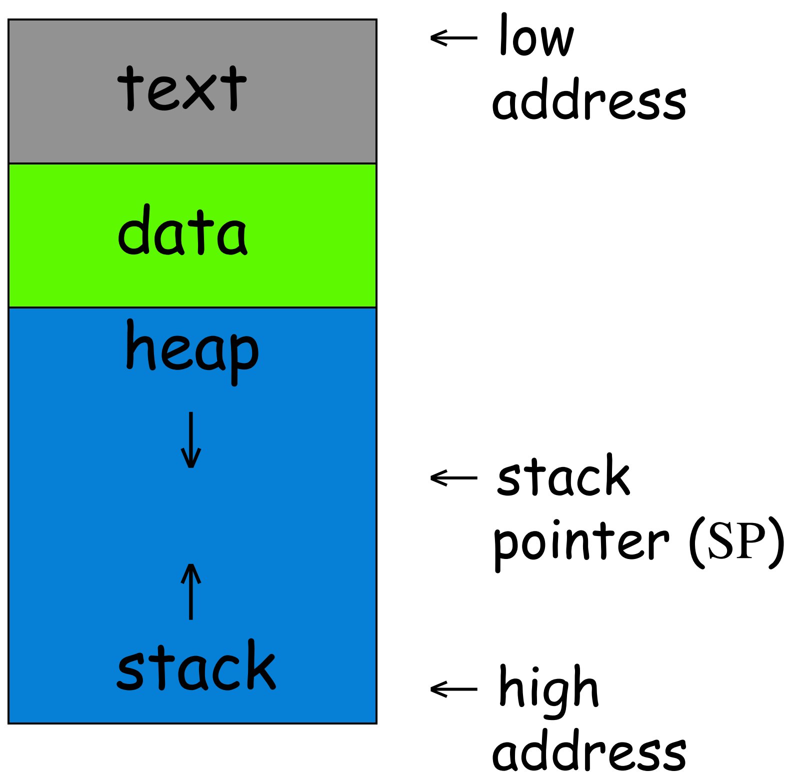
- Consider boolean flag for authentication
- Buffer overflow could overwrite flag allowing anyone to authenticate



- In some cases, Trudy need not be so lucky as in this example

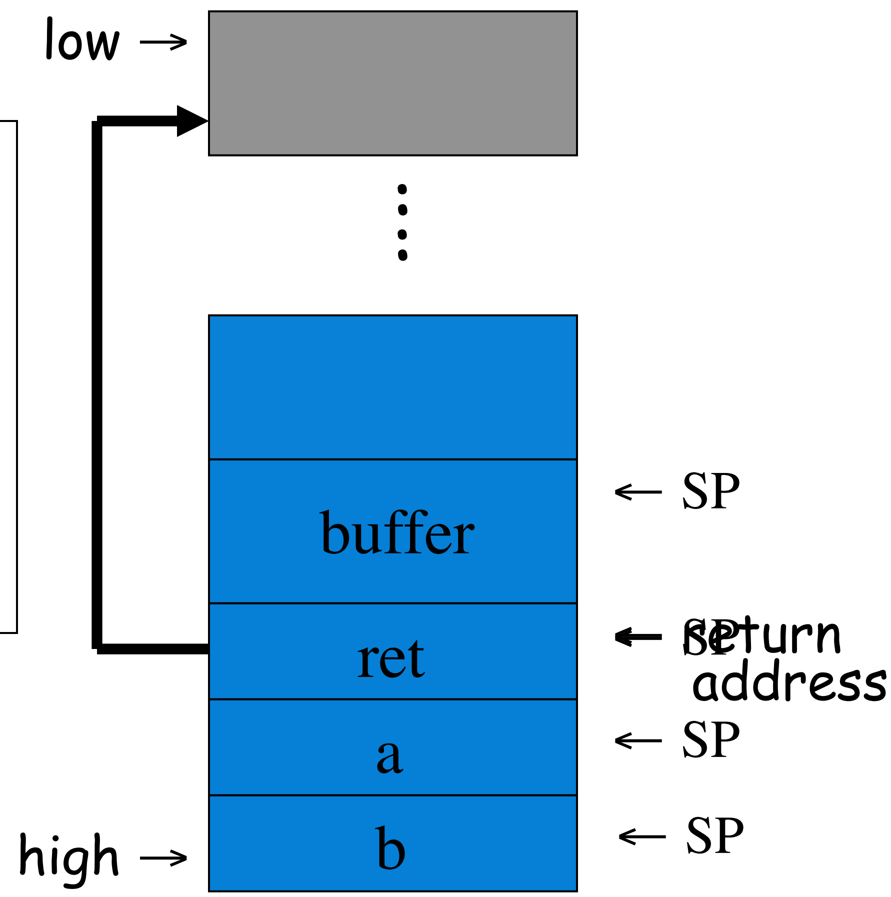
Memory Organization

- **Text** == code
- **Data** == static variables
- **Heap** == dynamic data
- **Stack** == "scratch paper"
  - Dynamic local variables
  - Parameters to functions
  - Return address



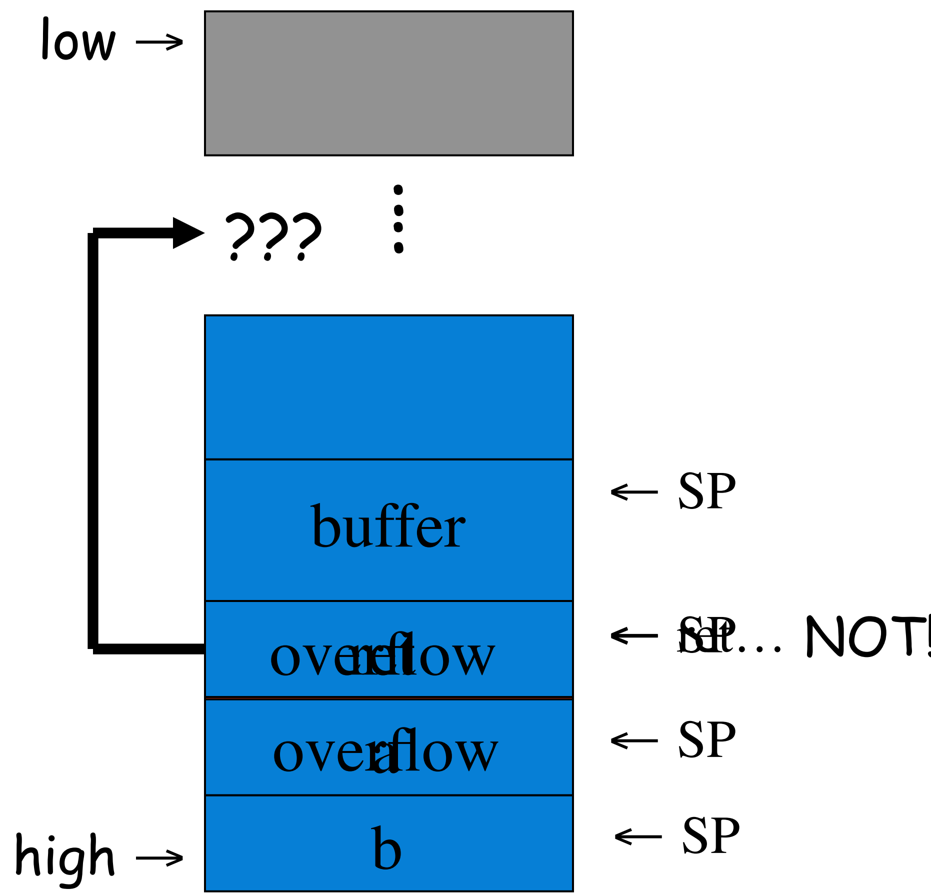
Simplified Stack Example

```
void func(int a, int b){
   char buffer[10];
}
void main(){
   func(1, 2);
}
```



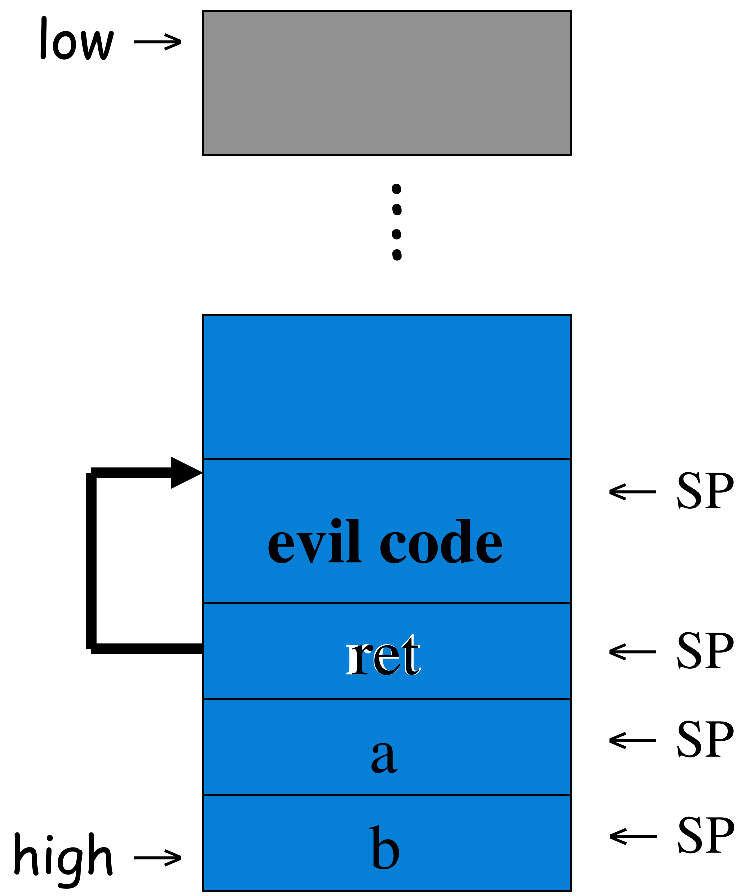
Smashing the Stack

- What happens if buffer overflows?
- Program "returns" to wrong location
- A crash is likely



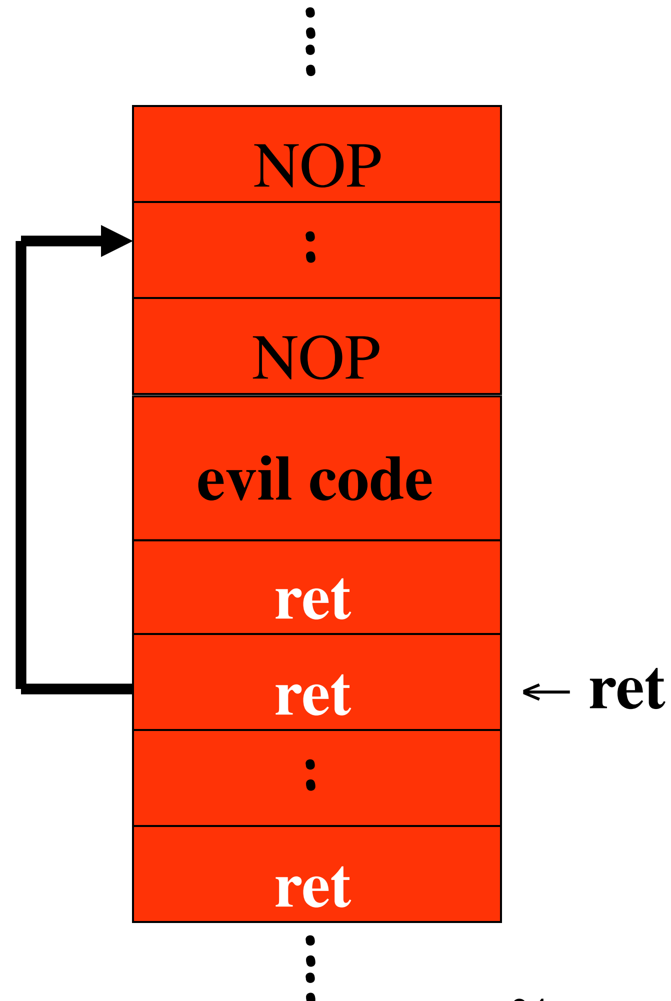
Smashing the Stack

- Trudy has a better idea…
- **Code injection**
- Trudy can run code of her choosing…
  - …on your machine



Smashing the Stack

- Trudy may not know…
  1) Address of evil code
  2) Location of **ret** on stack
- Solutions
  1) Precede evil code with NOP "landing pad"
  2) Insert **ret** many times

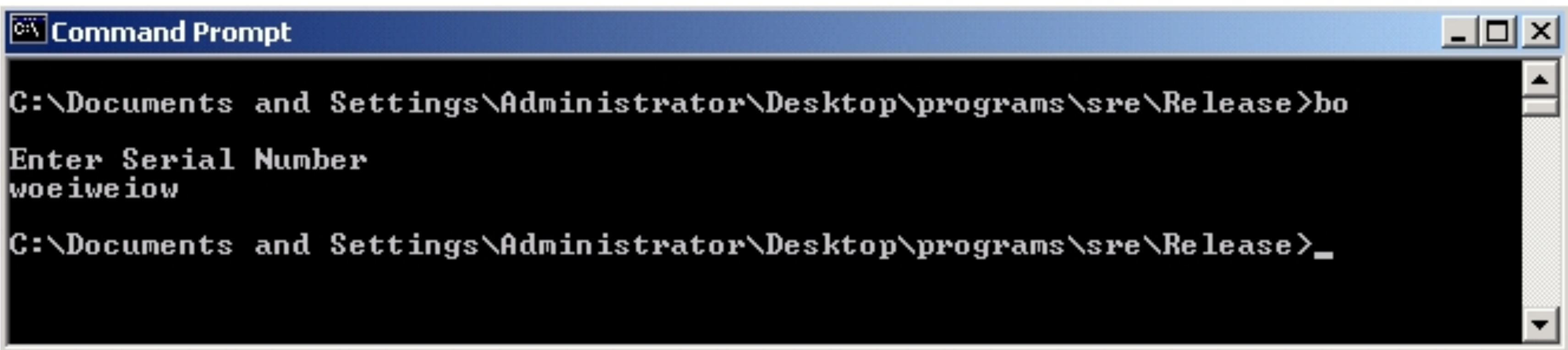


Stack Smashing Summary

- A buffer overflow must exist in the code
- Not all buffer overflows are exploitable
  - Things must align properly
- If exploitable, attacker can **inject code**
- Trial and error is likely required
  - Fear not, lots of help is available online
  - Smashing the Stack for Fun and Profit, Aleph One
- Stack smashing is "attack of the decade"
  - Regardless of the current decade
  - Also heap overflow, integer overflow, …

Stack Smashing Example

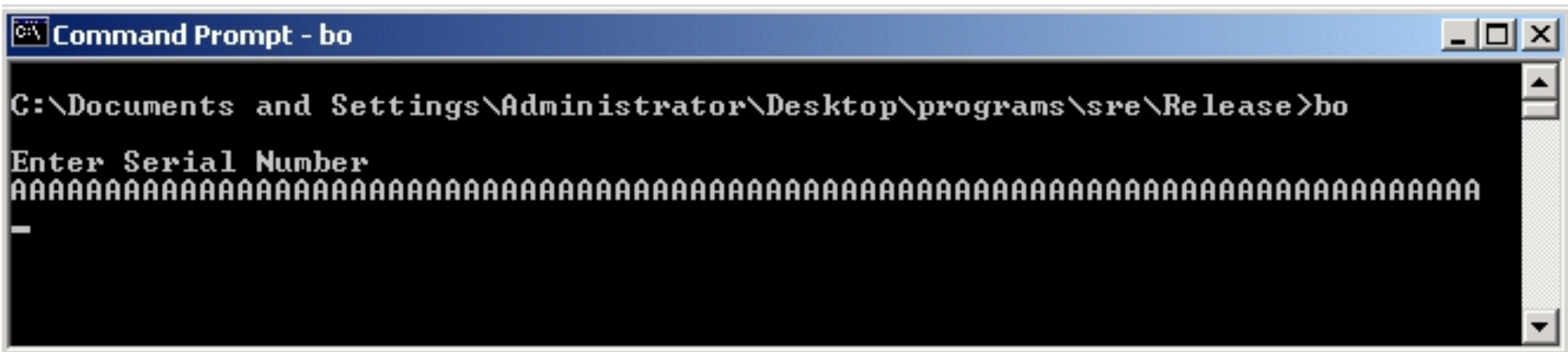
- Program asks for a serial number that the attacker does not know
- Attacker does **not** have source code
- Attacker does have the executable (exe)

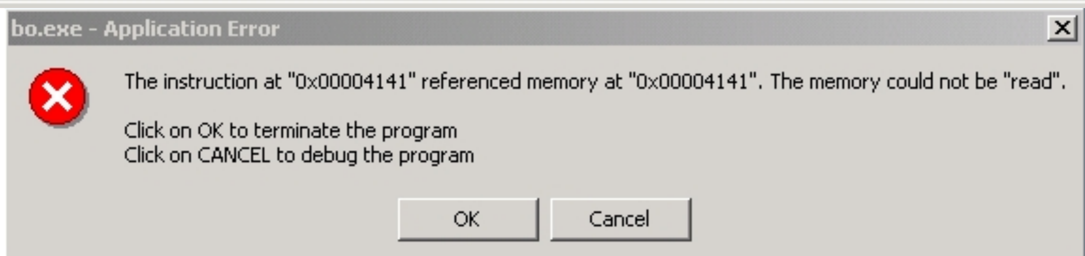


- Program quits on incorrect serial number

Buffer Overflow Present?

- By trial and error, attacker discovers apparent buffer overflow





- Note that 0x41 is ASCII for "A"
- Looks like **ret** overwritten by 2 bytes!

Disassemble Code

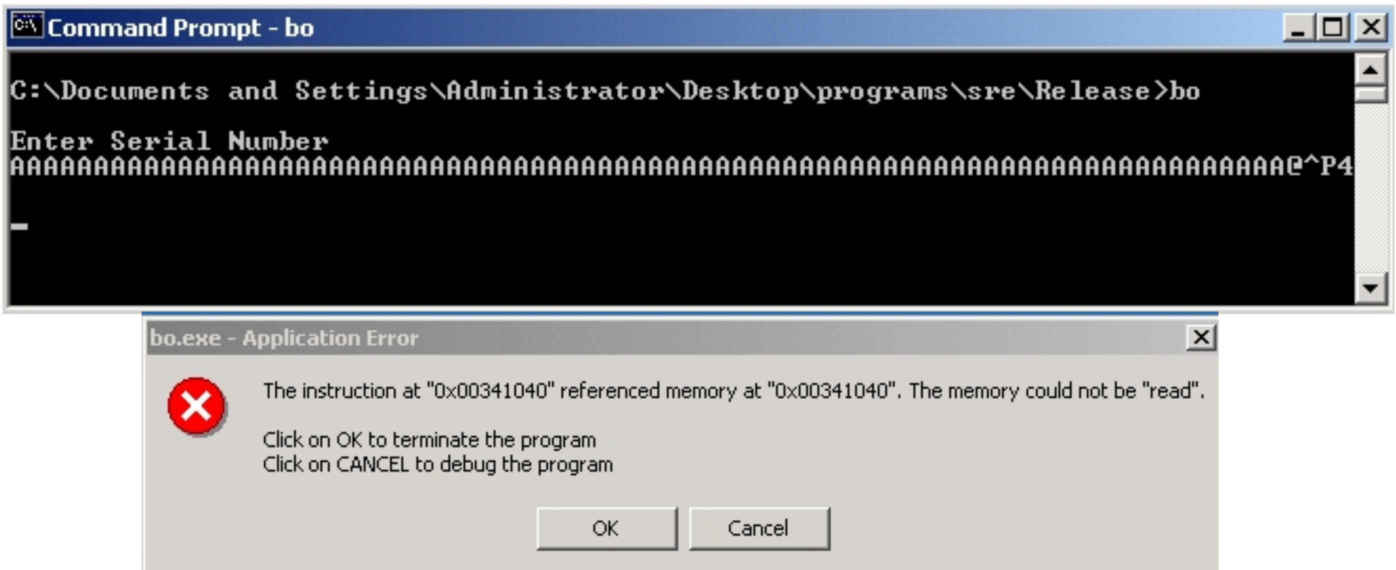
- Next, disassemble bo.exe to find

```
.text:00401000
.text:00401000                 sub     esp, 1Ch
.text:00401003                 push    offset aEnterSerialNum ; "\nEnter Serial Number\n"
.text:00401008                 call    sub_40109F
.text:0040100D                 lea     eax, [esp+20h+var_1C]
.text:00401011                 push    eax
.text:00401012                 push    offset aS       ; "%s"
.text:00401017                 call    sub_401088
.text:0040101C                 push    8
.text:0040101E                 lea     ecx, [esp+2Ch+var_1C]
.text:00401022                 push    offset aS123n456 ; "S123N456"
.text:00401027                 push    ecx
.text:00401028                 call    sub_401050
.text:0040102D                 add     esp, 18h
.text:00401030                 test    eax, eax
.text:00401032                 jnz     short loc_401041
.text:00401034                 push    offset aSerialNumberIs ; "Serial number is correct.\n"
.text:00401039                 call    sub_40109F
.text:0040103E                 add     esp, 4
```

- The goal is to exploit buffer overflow to jump to address 0x401034

Buffer Overflow Attack

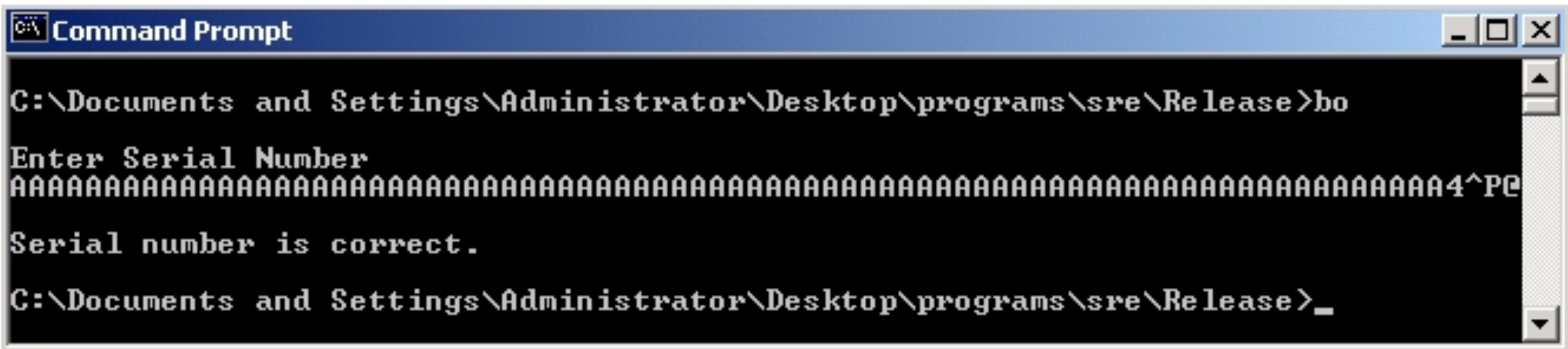
- Find that, in ASCII, 0x401034 is "@^P4"



- Byte order is reversed? Why?
- X86 processors are "little-endian"

Overflow Attack, Take 2

- Reverse the byte order to "4^P@" and…



- Success! We've bypassed serial number check by exploiting a buffer overflow
- What just happened?
  - Overwrote return address on the stack

Buffer Overflow

- Attacker did **not** require access to the source code
- Only tool used was a disassembler to determine address to jump to
- Find desired address by trial and error?
  - Necessary if attacker does not have exe
  - For example, a remote attack

Source Code

- Source code for buffer overflow example
- Flaw easily found by attacker…
- **…without access to source code!**

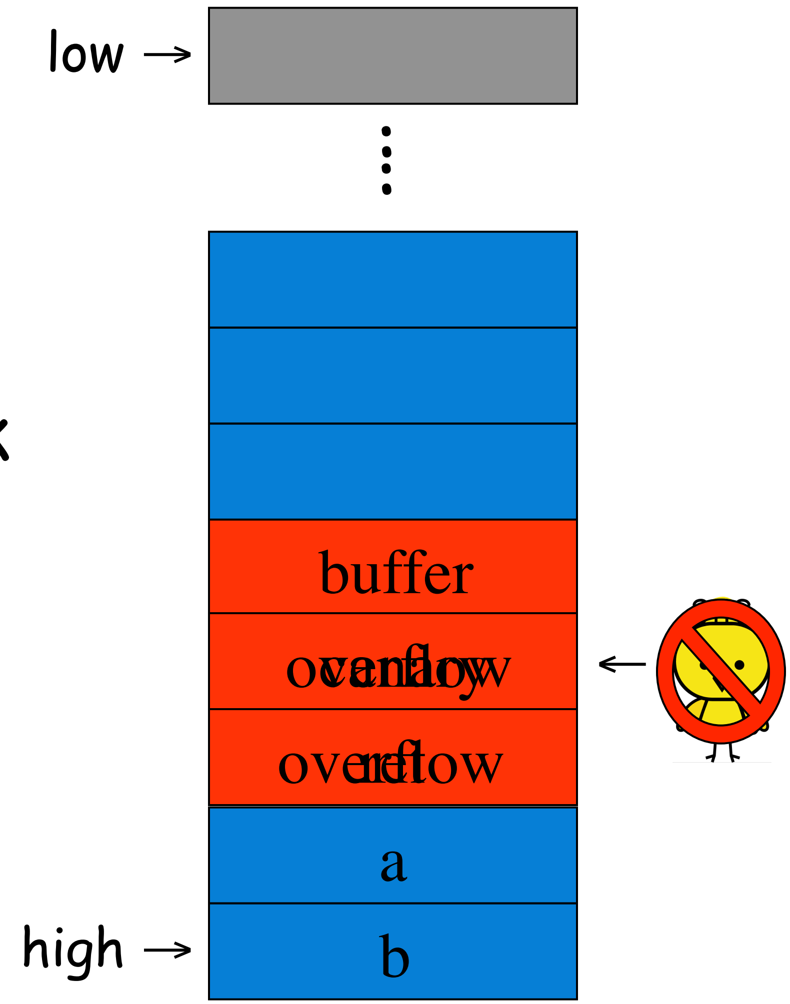
```
#include <stdio.h>
#include <string.h>

main()
{
    char in[75];

    printf("\nEnter Serial Number\n");

    scanf("%s", in);

    if(!strncmp(in, "S123N456", 8))
    {
        printf("Serial number is correct.\n");
    }
}
```

Stack Smashing Defenses

- Employ **non-executable stack**
  - "No execute" **NX bit** (if available)
  - Seems like the logical thing to do, but some real code executes on the stack (Java, for example)
- Use a **canary**
- Address space layout randomization (**ASLR**)
- Use **safe languages** (Java, C#)
- Use **safer C functions**
  - For unsafe functions, safer versions exist
  - For example, strncpy instead of strcpy

Stack Smashing Defenses

- **Canary**
  - Run-time stack check
  - Push canary onto stack
  - Canary value:
    - Constant 0x000aff0d
    - Or may depends on **ret**



Microsoft's Canary

- Microsoft added **buffer security check** feature to C++ with /GS compiler flag
  - Based on canary (or "security cookie")

Q: What to do when canary dies?

A: Check for user-supplied "handler"

- Handler shown to be subject to attack
  - Claim that attacker can specify handler code
  - If so, formerly "safe" buffer overflows become exploitable when /GS is used!

ASLR

- Address Space Layout Randomization
  - Randomize place where code loaded in memory
- Makes most buffer overflow attacks probabilistic
- Windows Vista uses 256 random layouts
  - So about 1/256 chance buffer overflow works?
- Similar thing in Mac OS X and other OSs
- Attacks against Microsoft's ASLR do exist
  - Possible to "de-randomize"

Buffer Overflow

- A major security threat yesterday, today, and tomorrow
- The good news?
- It <u>is</u> possible to reduced overflow attacks
  - Safe languages, NX bit, ASLR, education, etc.
- The bad news?
- Buffer overflows will exist for a long time
  - Legacy code, bad development practices, etc.

Incomplete Mediation

Input Validation

- Consider: `strcpy(buffer, argv[1])`
- A buffer overflow occurs if

  `len(buffer) < len(argv[1])`
- Software must **validate** the input by checking the length of `argv[1]`
- Failure to do so is an example of a more general problem: **incomplete mediation**

Input Validation

- Consider web form data
- Suppose input is validated on client
- For example, the following is valid

```
http://www.things.com/orders/
  final&custID=112&num=55A&qty=20&price=10&shi
  pping=5&total=205
```

- Suppose input is not checked on server
  - Why bother since input checked on client?
  - Then attacker could send http message

```
http://www.things.com/orders/
  final&custID=112&num=55A&qty=20&price=10&shi
  pping=5&total=25
```

Incomplete Mediation

- Linux kernel
  - Research has revealed many buffer overflows
  - Many of these are due to incomplete mediation
- Linux kernel is "good" software since
  - Open-source
  - Kernel — written by coding gurus
- Tools exist to help find such problems
  - But incomplete mediation errors can be subtle
  - And tools useful to attackers too!

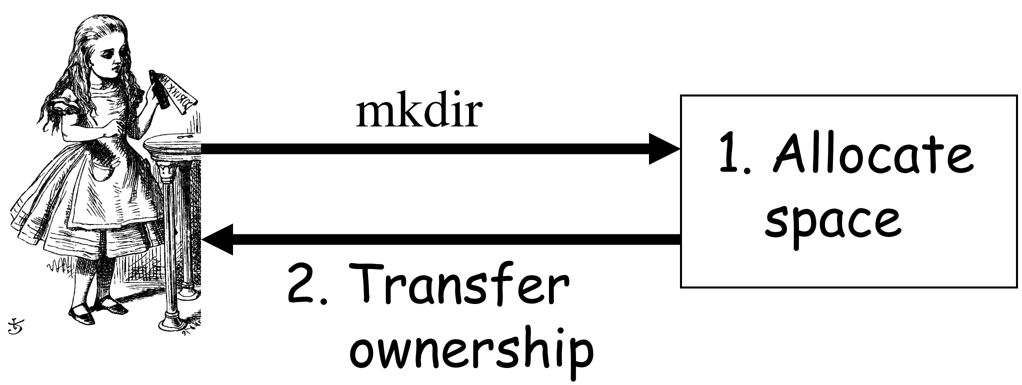
Race Conditions

Race Condition

- Security processes should be **atomic**
  - Occur "all at once"
- Race conditions can arise when security-critical process occurs in stages
- Attacker makes change between stages
  - Often, between stage that gives authorization, but before stage that transfers ownership
- Example: Unix mkdir

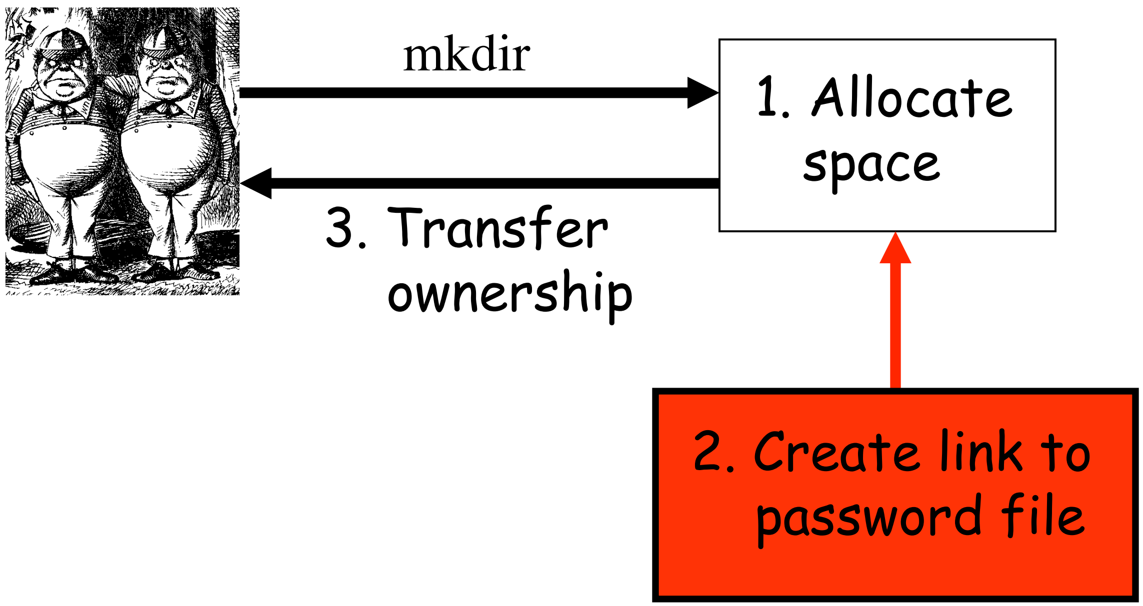
mkdir Race Condition

- mkdir creates new directory
- How mkdir is supposed to work



mkdir Attack

- The mkdir **race condition**



- Not really a "race"
  - But attacker's timing is critical

Race Conditions

- Race conditions are common
- Race conditions may be more prevalent than buffer overflows
- But race conditions harder to exploit
  - Buffer overflow is “low hanging fruit” today
- To prevent race conditions, make security-critical processes atomic
  - Occur all at once, not in stages
  - Not always easy to accomplish in practice

Buffer Overflow Lab