Information Security
CS3002

Lecture 13
2nd October 2024

Dr. Rana Asif Rehman
Email: r.asif@lhr.nu.edu.pk

Control Hijacking

Source: web.nvd.nist.gov

Buffer Overflow

NIST's Definition: Buffer overflow

"A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system."

Control Hijacking - Buffer Overflow

- **A buffer overflow**, or **buffer** overrun, is an anomaly where a program, while writing data to a **buffer**, overruns the **buffer's** boundary and overwrites adjacent memory locations.
- **Causes:**
  - Systems software is often written in C(OS, compilers, databases, network servers)
  - C is high-level assembly so..
    - Exposes raw pointers to memory
    - Does not perform bound checking on arrays.
  - Attack also leverages architectural knowledge of how x86 code works.
    - The Direction that stack grows.
    - Layout of stack variables.

- Buffer overflows are important in sense as it matters which process you are hijacking. If you are getting root process hijacked then that's a real disaster.

Buffer Overflow Basics

- Caused by programming error
- Allows more data to be stored than capacity available in a fixed sized buffer
  - buffer can be on stack, heap, global data
- Overwriting adjacent memory locations
  - corruption of program data
  - unexpected transfer of control
  - memory access violation
  - execution of code chosen by attacker

Buffer Overflow Attacks

- To exploit a buffer overflow an attacker
  - must identify a buffer overflow vulnerability in some program
    - Inspection (program source) , tracing execution (oversized input), fuzzing tools
  - understand how buffer is stored in memory and determine potential for corruption (adjacent memory) and altering the flow of execution

A Little Programming Language

- At machine level, all data is an array of bytes (processor's registers or in memory)
  - interpretation depends on instructions used
- Modern high-level languages have a strong notion of variable types and valid permissible operations on them
  - not vulnerable to buffer overflows
  - does incur overhead (resources use both compile/run-time to impose checks on buffer limits), some limits on use (access to some instructions and hardware resources is lost)
- C and related languages have high-level control structures, but allow direct access to memory
  - hence are vulnerable to buffer overflow
  - have a large legacy of widely used, unsafe, and hence vulnerable code

Control Hijacking - Buffer Overflow

Why is an assumption about **data length** dangerous?

- Computer programs organise internal data values in ***variables*** stored in memory

Reserved **50** characters for email

PC Memory

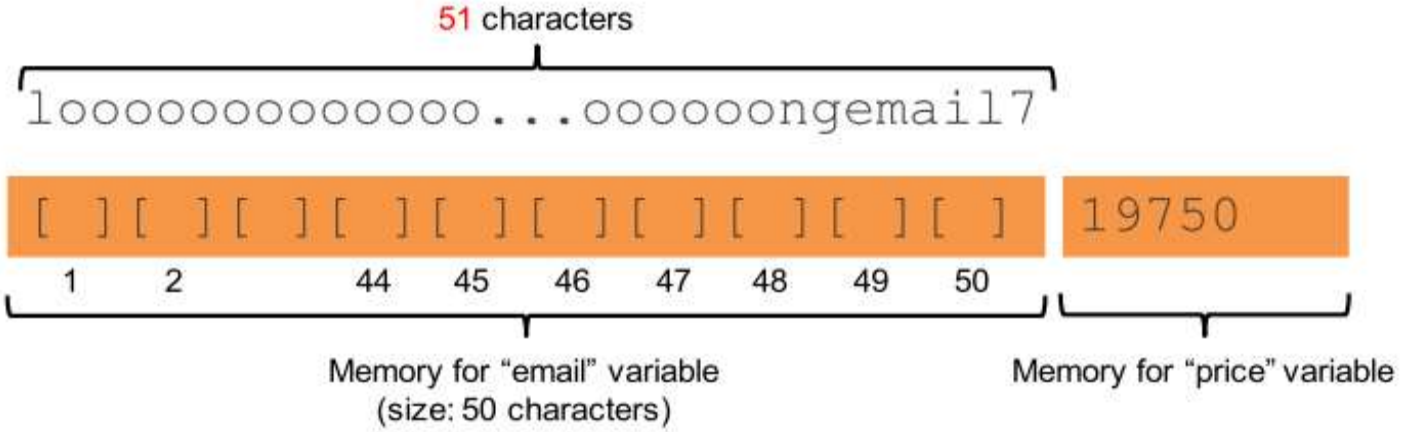
Program "Car configurator"

Variables: Red | Yes | loooooooooooooooooongemail

Another program...

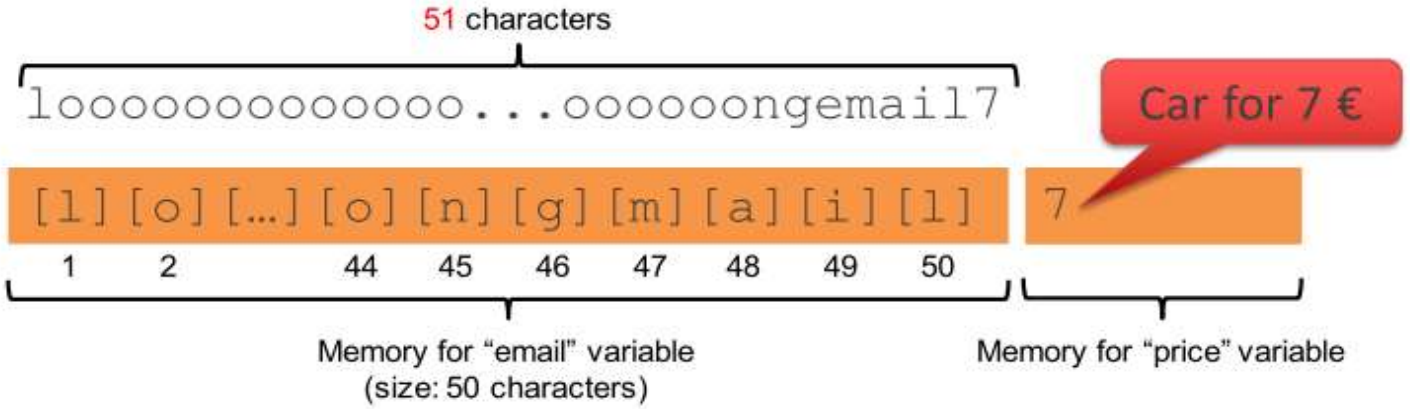
Control Hijacking - Buffer Overflow

- The car configurator example allows us to overwrite the associated price using a very long email address

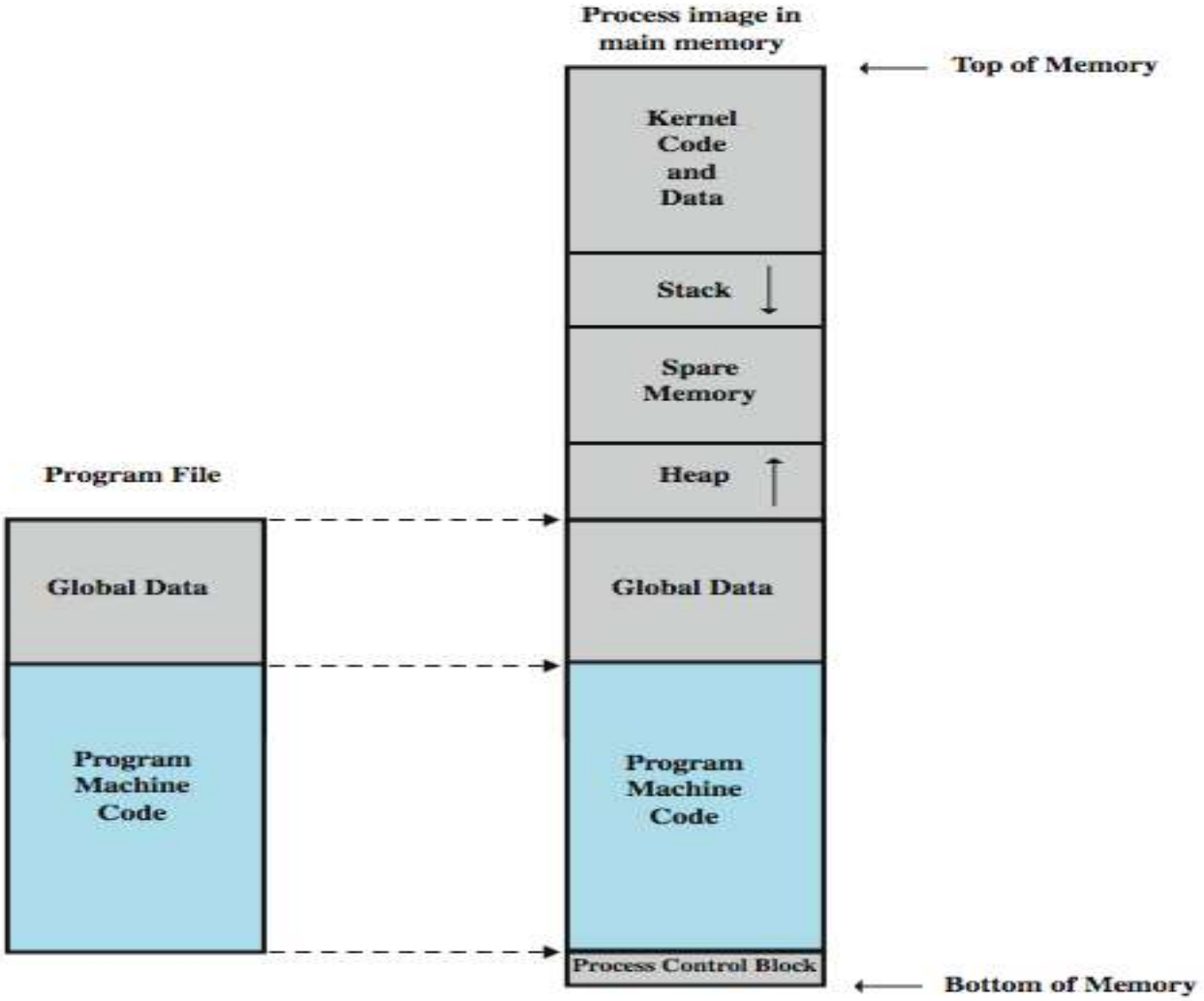


Control Hijacking - Buffer Overflow

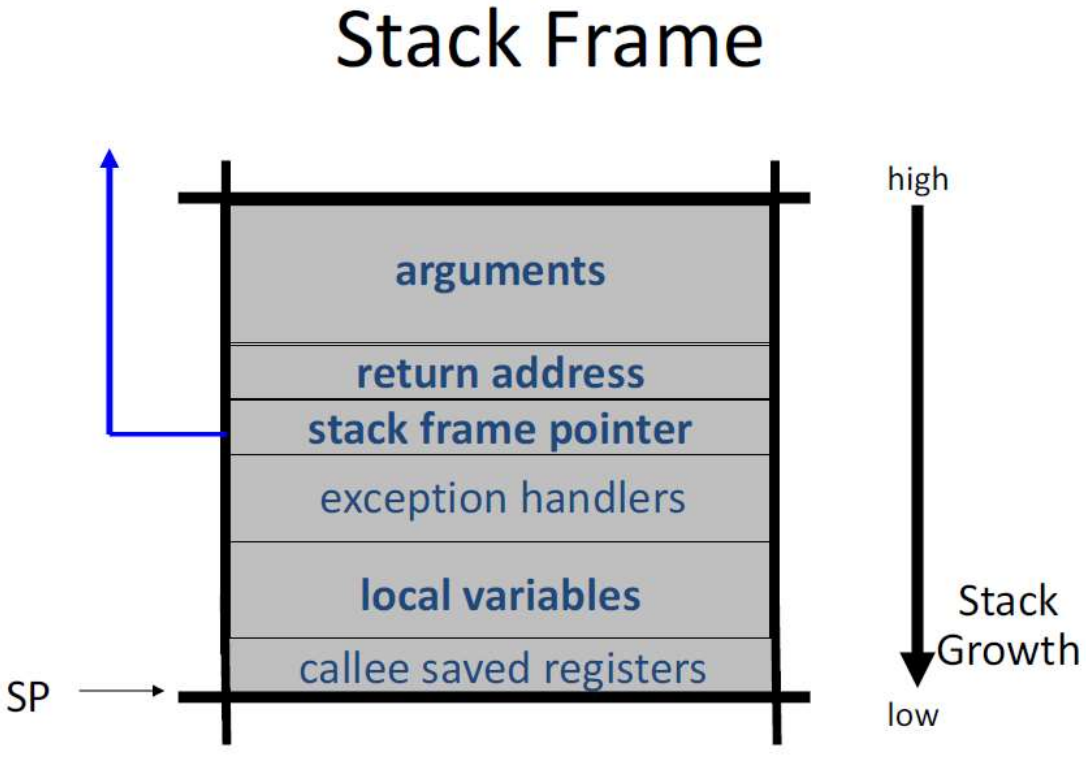
- The car configurator example allows us to overwrite the associated price using a very long email address



Control Hijacking - Buffer Overflow



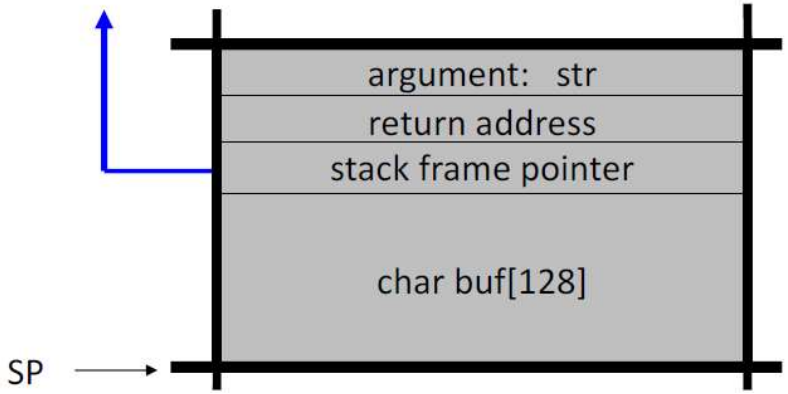
Control Hijacking - Buffer Overflow



Control Hijacking - Buffer Overflow

Suppose a web server contains a function:

When func() is called stack looks like:

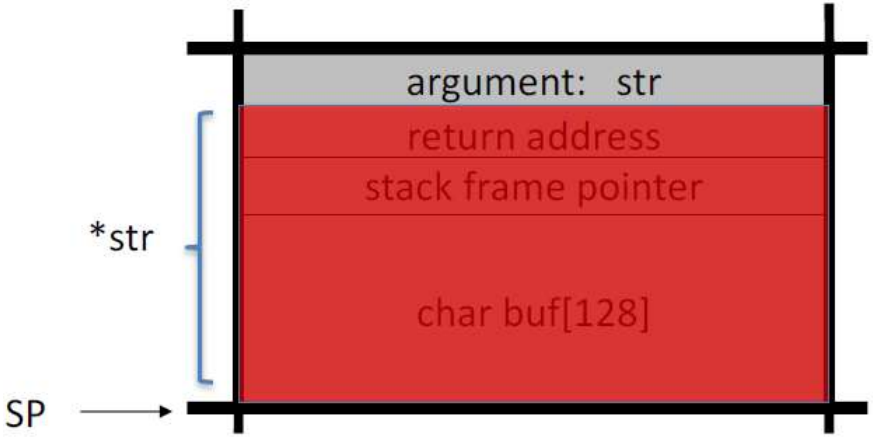



```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    do-something(buf);
}
```

Control Hijacking - Buffer Overflow

What if `*str` is 136 bytes long?

After `strcpy`:




```
void func(char *str) {
    char buf[128];
    strcpy(buf, str);
    do-something(buf);
}
```

Problem:

no length checking in strcpy()

Control Hijacking - Buffer Overflow

Many Unsafe Functions that lead to Buffer Overflows:

strcpy (char *dest, const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf (const char *format, …) and many more.

- "Safe" libc versions strncpy(), strncat() are misleading
  - e.g. strncpy() may leave string unterminated.
- Windows C run time (CRT):
  - strcpy_s (*dest, DestSize, *src): ensures proper termination

Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:
  `$ prog ABCD`

Control flow | Source code of `prog` | Stack during execution (simplified)

```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

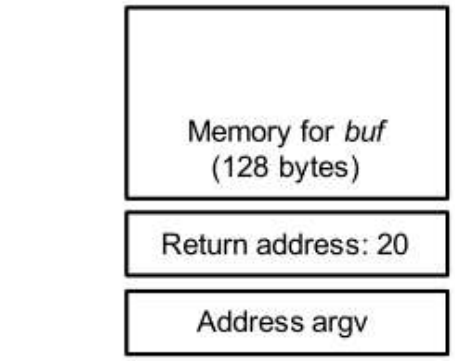
Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:
  `$ prog ABCD`

Control flow | Source code of `prog` | Stack during execution (simplified)

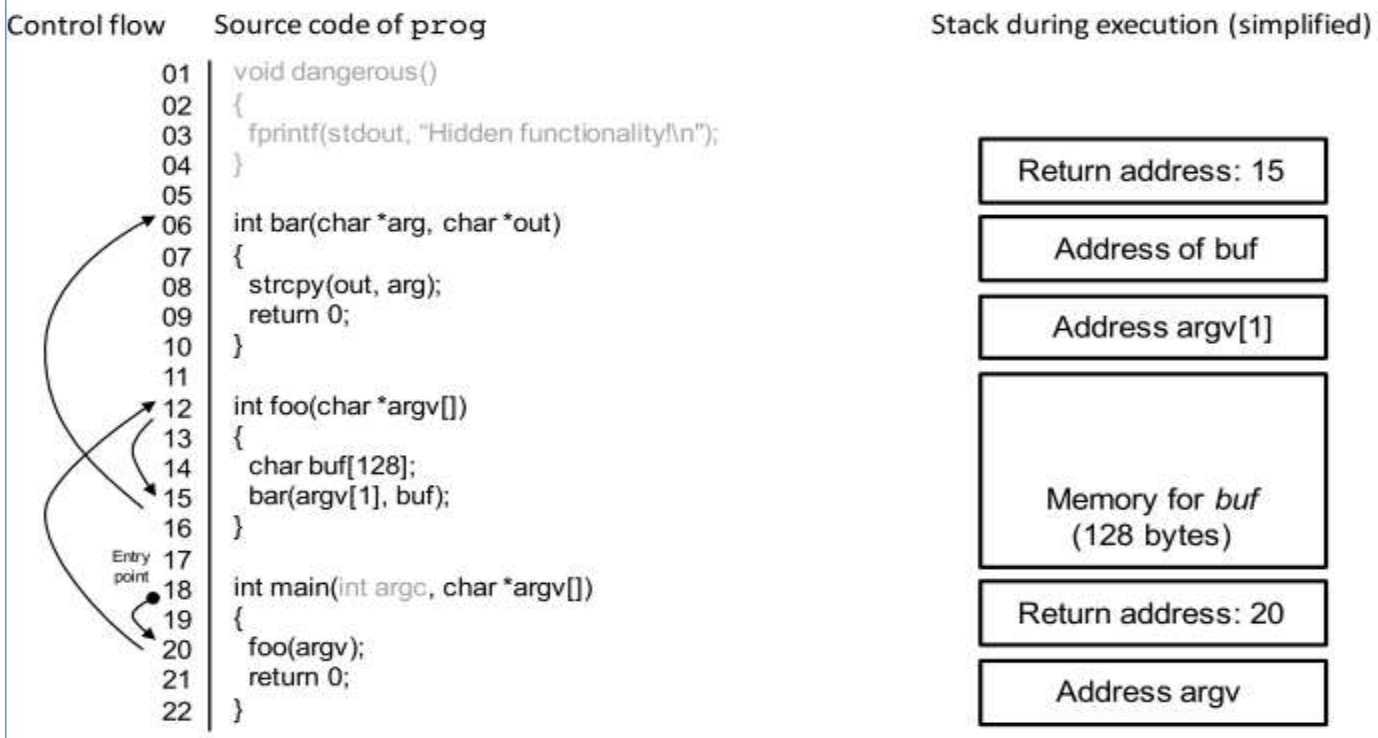
```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

Entry point



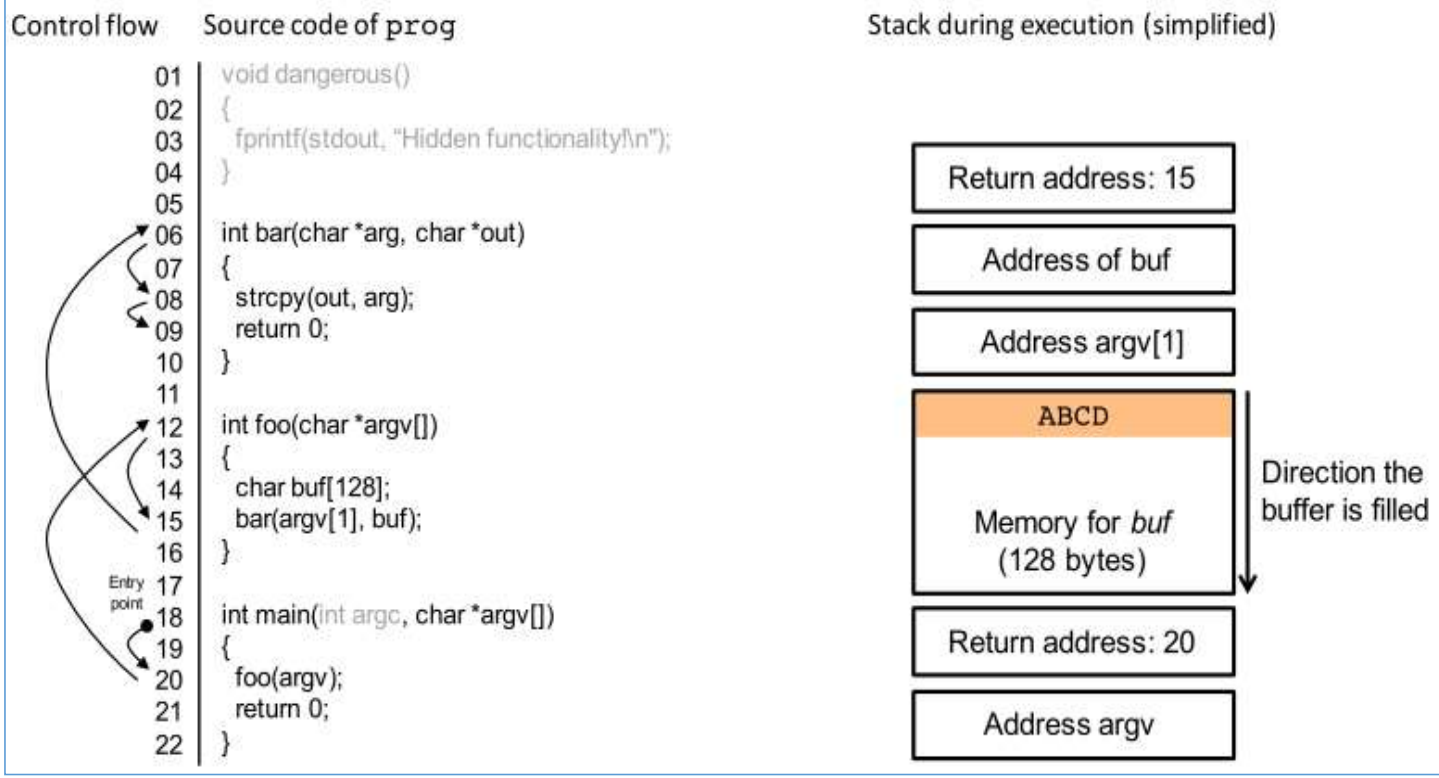
Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:
  `$ prog ABCD`



Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:
  `$ prog ABCD`



Control Hijacking - Buffer Overflow
Control Flow with Stack

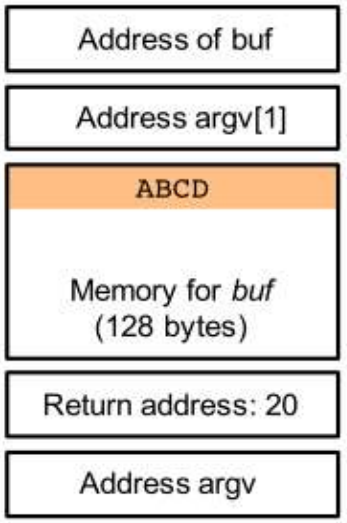
- Calling a simple program on shell:
  `$ prog ABCD`

Control flow | Source code of `prog`

```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

Current instruction → 09

Stack during execution (simplified)



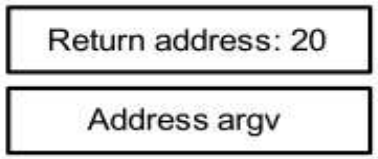
Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:
  `$ prog ABCD`

Control flow | Source code of `prog` | Stack during execution (simplified)

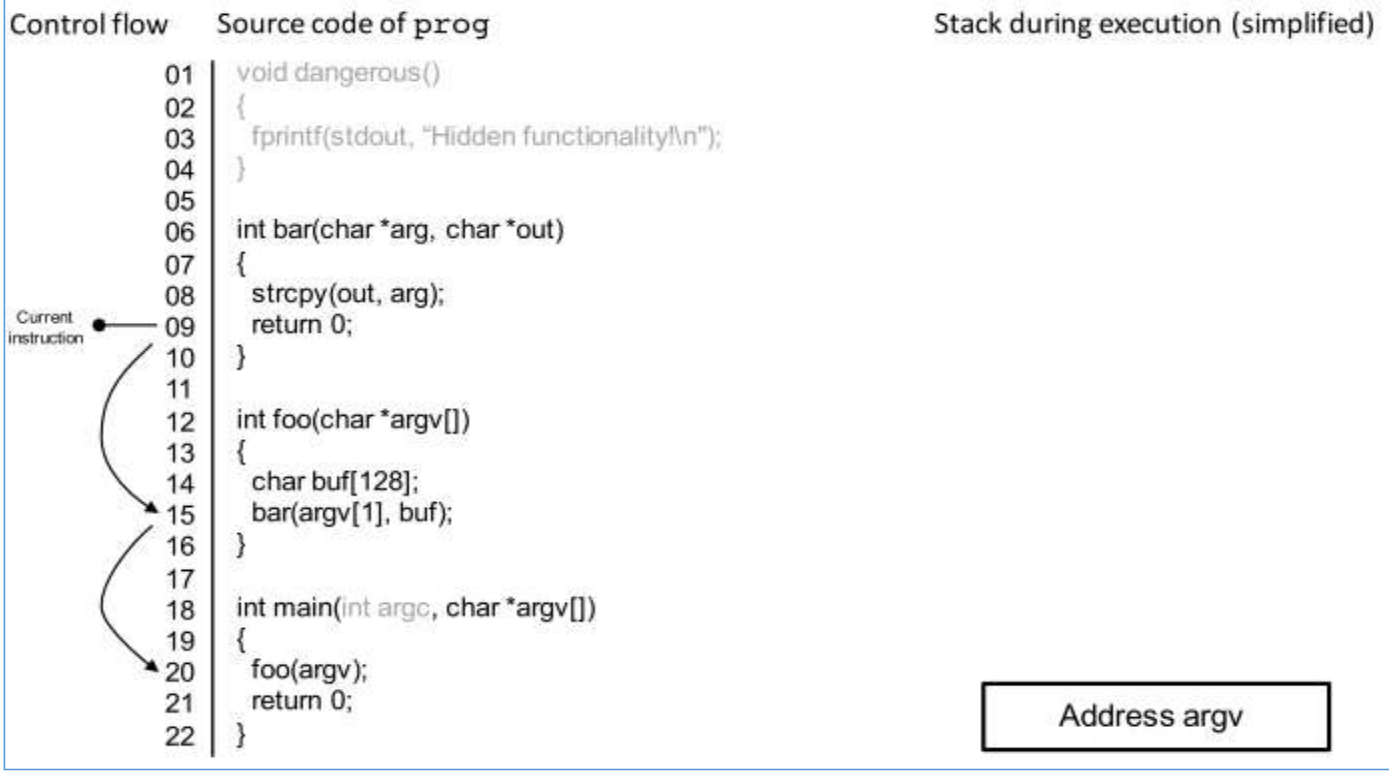
```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

Current instruction → line 09



Control Hijacking - Buffer Overflow
Control Flow with Stack

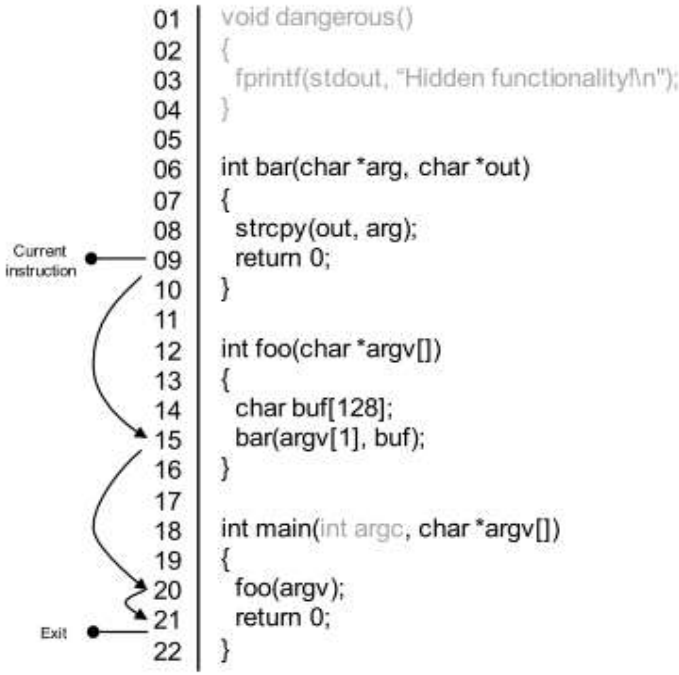
- Calling a simple program on shell:
  `$ prog ABCD`



Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:
  `$ prog ABCD`

Control flow | Source code of `prog` | Stack during execution (simplified)



Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:

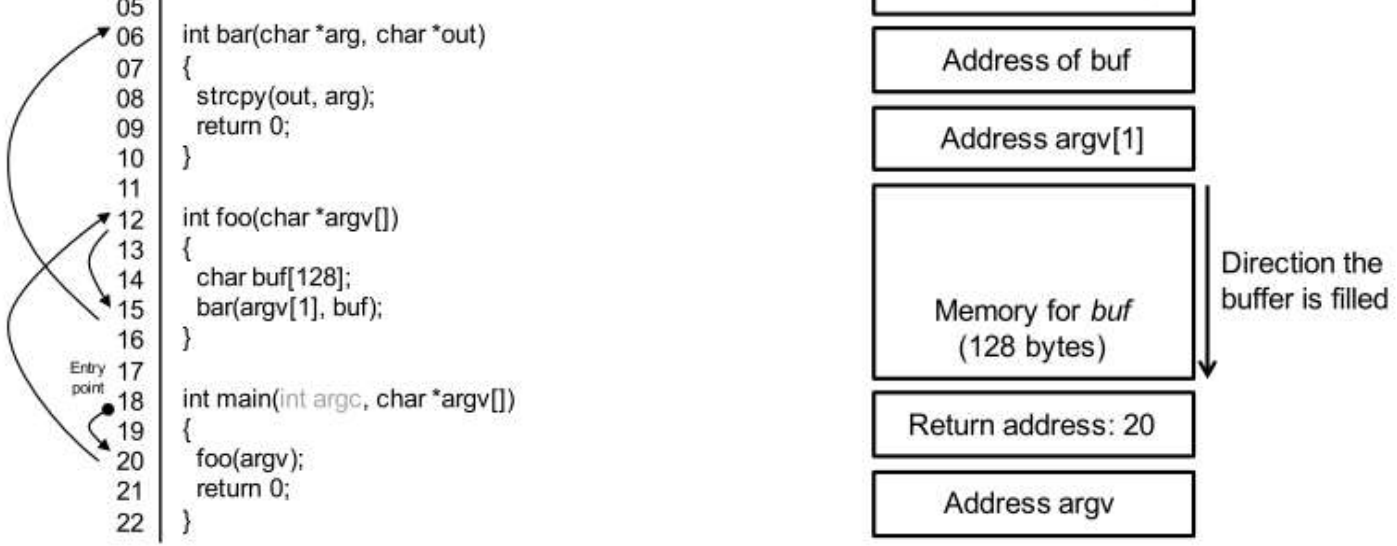
```
$ prog AAAAAAAAAAAAAAAA...AAAAAAAAAA01
```

128 bytes

Control flow | Source code of `prog` | Stack during execution (simplified)

```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

Entry point



Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:

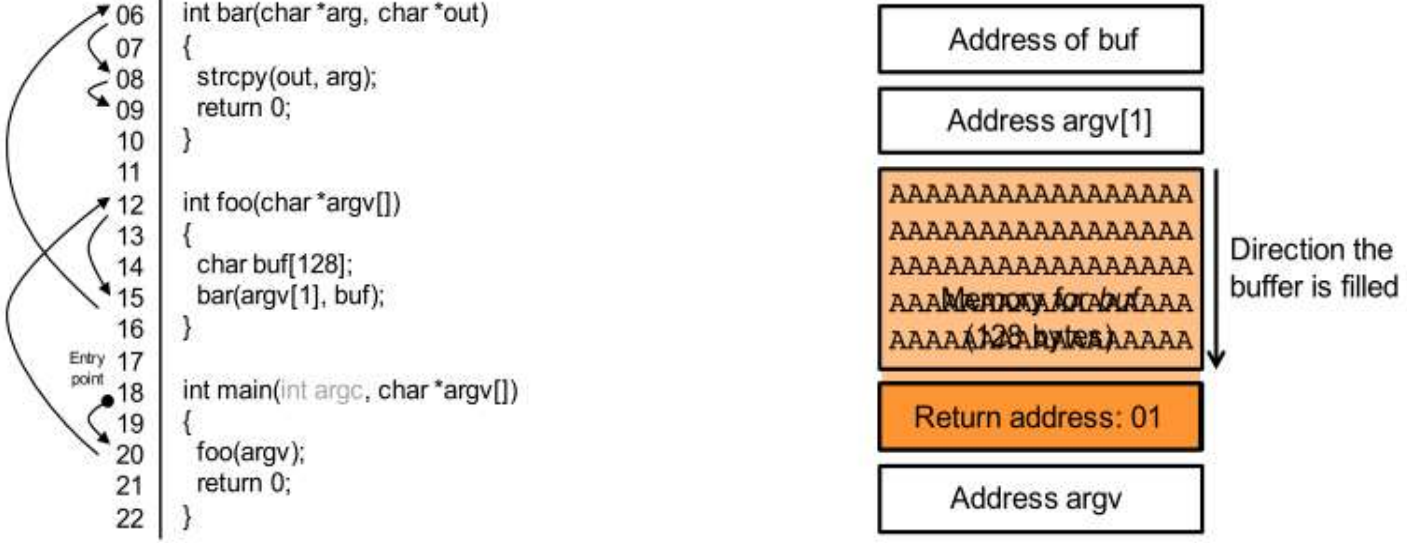
```
$ prog AAAAAAAAAAAAAAAA...AAAAAAAAAA01
```

128 bytes

Control flow | Source code of `prog` | Stack during execution (simplified)

```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

Entry point



Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:

```
$ prog AAAAAAAAAAAAAAAA...AAAAAAAAAA01
```

128 bytes

Control flow | Source code of `prog` | Stack during execution (simplified)

```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

Current instruction → line 09

Stack during execution (simplified):

- Return address: 15
- Address of buf
- Address argv[1]
- AAAAAAAAAAAAAAAA … (Memory for buf (128 bytes))
- Return address: 01
- Address argv

Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:

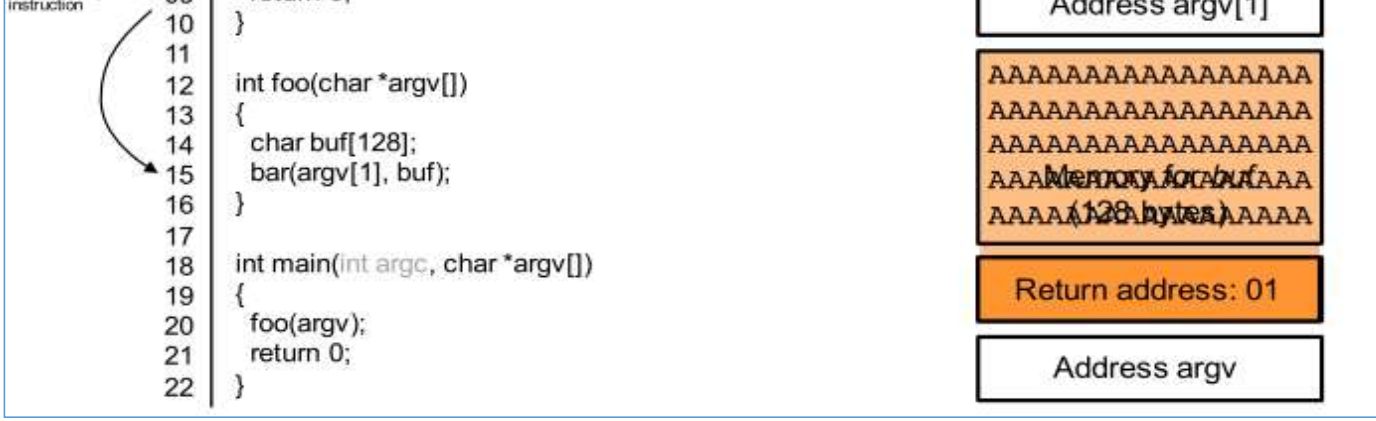
```
$ prog AAAAAAAAAAAAAAAA...AAAAAAAAAA01
```

128 bytes

Control flow | Source code of `prog` | Stack during execution (simplified)

```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

Current instruction → 09



Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:

```
$ prog AAAAAAAAAAAAAAAAA...AAAAAAAAAA01
```

128 bytes

Control flow | Source code of `prog` | Stack during execution (simplified)

```
01  void dangerous()
02  {
03    fprintf(stdout, "Hidden functionality!\n");
04  }
05
06  int bar(char *arg, char *out)
07  {
08    strcpy(out, arg);
09    return 0;
10  }
11
12  int foo(char *argv[])
13  {
14    char buf[128];
15    bar(argv[1], buf);
16  }
17
18  int main(int argc, char *argv[])
19  {
20    foo(argv);
21    return 0;
22  }
```

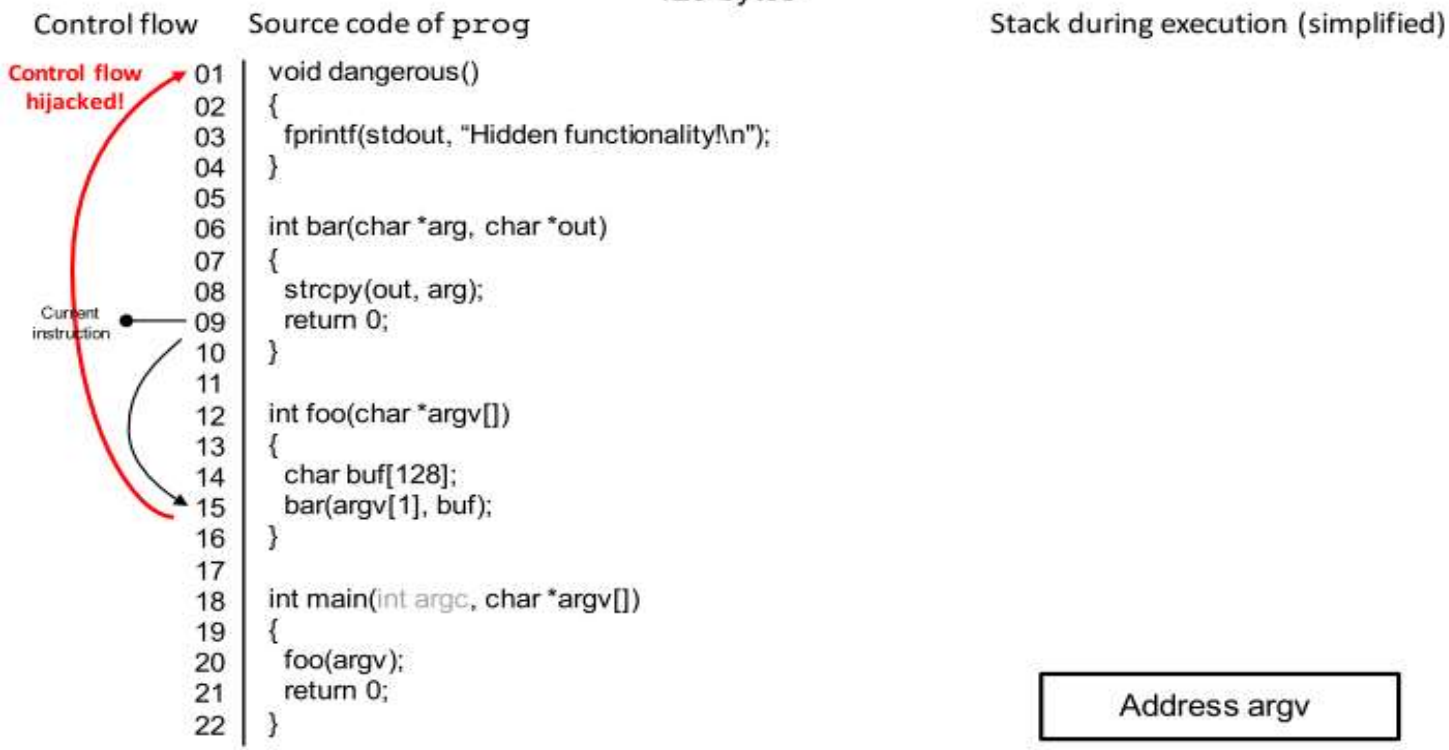
Current instruction → 09

Control Hijacking - Buffer Overflow
Control Flow with Stack

- Calling a simple program on shell:

```
$ prog AAAAAAAAAAAAAAAAA...AAAAAAAAAA01
```

128 bytes



What can attacker do now?

Jumps wherever he wants

- Attacker now is running the code with the privileges of process that it's hijacked.
  - Root_user, admin, super_user
  - Send spam mails by overtaking mail server
  - Defeat the firewall / Private Network compromised / Trust Broken

Real Life Examples

- Buffer overflow vulnerabilities were exploited by the first major attack on the Internet. Known as the Morris worm, this attack infected more than 60,000 machines (10%) and shut down much of the Internet for several days in 1988.
- Source: Carolyn Duffy Marsan, Morris Worm Turns 20: Look what it's Done, Network World, October 30, 2008 http://www.techworld.com.au/article/265692/morris_worm_turns_20_look_what_it_done/

Robert Tappan Morris
https://pdos.csail.mit.edu/~rtm/

Real Life Examples (cont.)

- A buffer overflow in a 2004 version of AOL's AIM instant-messaging software exposed users to buffer overflow vulnerabilities. If a user posted a URL in their "I'm away" message, any of his or her friends who clicked on that link might be vulnerable to attack. AOL's response was to suggest that users update to a new version that would fix the bug.
- Source: Paul Roberts "AOL IM 'Away' message flaw deemed critical", Infoworld, August 9, 2004 http://www.infoworld.com/article/04/08/09/HNaolimflaw_1.html

Adversary Motivations

- Use any privileges of the process
- Often leverage overflow to gain easier access to the system.
  - Originally on UNIX, run shell /bin/sh (shell code)
- If the process running as root or administrator , can do anything
- Even if not can send spam emails, read files.
- Can attack other machines behind a firewall.

Buffer Overflow Countermeasures

Buffer Overflow Defenses

- Buffer overflows are widely exploited
- Large amount of vulnerable code in use
  - despite cause and countermeasures known
- Two broad defense approaches
  - compile-time - harden new programs
  - run-time - handle attacks on existing programs

Compile-Time Defenses: Programming Language

- Use a modern high-level languages with strong typing
  - not vulnerable to buffer overflow
  - compiler enforces range checks and permissible operations on variables
- Do have cost in resource use
- And restrictions on access to hardware
  - so still need some code in C like languages i.e. device drivers

Compile-Time Defenses: Safe Coding Techniques

- If using potentially unsafe languages e.g. C
- Programmer must explicitly write safe code
  - by design with new code
  - ***extensive after code review*** of existing code, (e.g., OpenBSD)
- Buffer overflow safety a subset of general safe coding techniques
- Allow for graceful failure ***(know how things may go wrong)***
  - check for sufficient space in any buffer

Compile-Time Defenses: Language Extension, Safe Libraries

- Proposals for safety extensions (library replacements) to C
  - Compilers automatically insert range checks om unsafe array/pointer references
  - performance penalties
  - must re-compiled programs with special modified compiler

- Several safer standard library variants
  - new functions, e.g. strlcpy()
  - safer re-implementation of standard functions as a dynamic library, e.g. Libsafe (it includes additional checks to ensure that the copy operations do not extend beyond the local variable space in the stack frame)

Compile-Time Defenses: Stack Protection

- Stackgaurd: add function entry and exit code to check stack for signs of corruption
  - Use random canary
  - e.g. Stackguard, Win/GS, GCC
  - check for overwrite between local variables and saved frame pointer and return address
  - abort program if change found
  - issues: recompilation, debugger support
- Or save/check safe copy of return address (in a safe, non-corruptible memory area), e.g. Stackshield, RAD (return Address Defender), do not alter the stack frame

Run-Time Defenses: Non Executable Address Space

- Many BO attacks copy machine code into buffer and transfer ctrl to it
- Use virtual memory support to make some regions of memory non-executable (to avoid exec of attacker's code)
  - e.g. stack, heap, global data
  - Read/Write data but not executable
  - need h/w support in MMU
  - long existed on SPARC/Solaris systems
  - recent on x86 Linux/Unix/Windows systems
- Issues: support for executable stack code (nested functions in C, Linux signal handlers)

Run-Time Defenses: Address Space Randomization

- Manipulate location of key data structures
  - stack, heap, global data: change address by 1 MB
  - using random shift for each process
  - have large address range on modern systems means wasting some has negligible impact

- Randomize location of heap buffers and location of standard library functions

Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory (or between stack frames)
  - flagged in MMU (mem mgmt unit) as illegal addresses
  - any access aborts process
- This can prevent buffer overflow attacks, typically of global data, which attempt to overwrite adjacent regions in the processes address space, such as the global offset table.
- Can even place between stack frames and heap buffers
  - at execution time and space cost