Malicious Code for Fun and Profit

Mihai Christodorescu

mihai@cs.wisc.edu

10 March 2005

What is Malicious Code?

Viruses, worms, trojans, ...

Code that breaks your security policy.

Characteristics
- Attack vector
- Payload
- Spreading algorithm

Outline

- Attack Vectors
- Payloads
- Spreading Algorithms
- Case Studies

Attack Vectors

- Social engineering

  "Make them want to run it."

- Vulnerability exploitation

  "Force your way into the system."

- Piggybacking

  "Make it run when other programs run."

Social Engineering

- Suggest to user that the executable is:
  - A game.
  - A desirable picture/movie.
  - An important document.
  - A security update from Microsoft.
  - A security update from the IT department.
- Spoofing the sender helps.

Outline

- Attack Vectors:
  - Social Engineering
  - Vulnerability Exploitation
  - Piggybacking
- Payloads
- Spreading Algorithms
- Case Studies

Vulnerability Exploitation

- Make use of flaws in software input handling.
- Sample techniques:
  - Buffer overflow attacks.
  - Format string attacks.
  - Return-to-libc attacks.
  - SQL injection attacks.

Basic Principles

A buffer overflow occurs when data is stored past the boundaries of an array or a string.

The additional data now overwrites nearby program variables.

Result:

Attacker controls or takes over a currently running process.

Example

Expected input: \\hostname\path

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```

Example

Expected input: \\hostname\path

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );
```

process_request("\\tux12\usr\foo.txt"); ⇒ ✓ OK

Example

Expected input: \\hostname\path

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );
```

process_request("\\tux12\usr\foo.txt"); ⇒ ✓ OK

process_request("\\aaabbbcccdddeeefffgggghhh\bar"); ⇒ ✘ BAD

Program Stack

A stack frame per procedure call.

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```

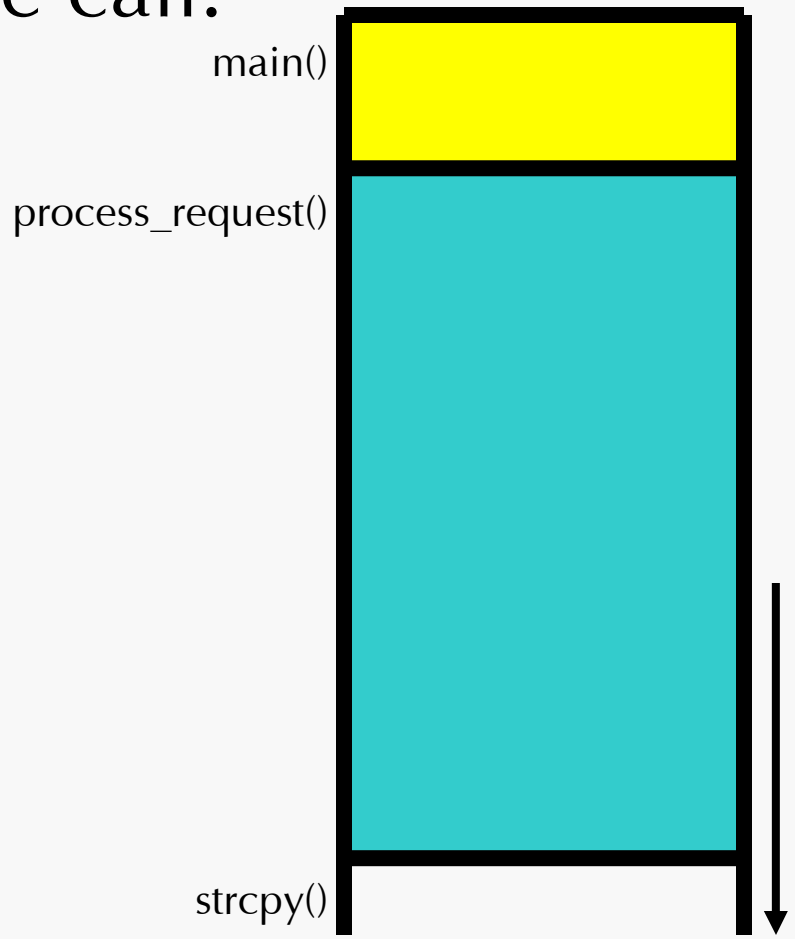
Program Stack

A stack frame per procedure call.

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



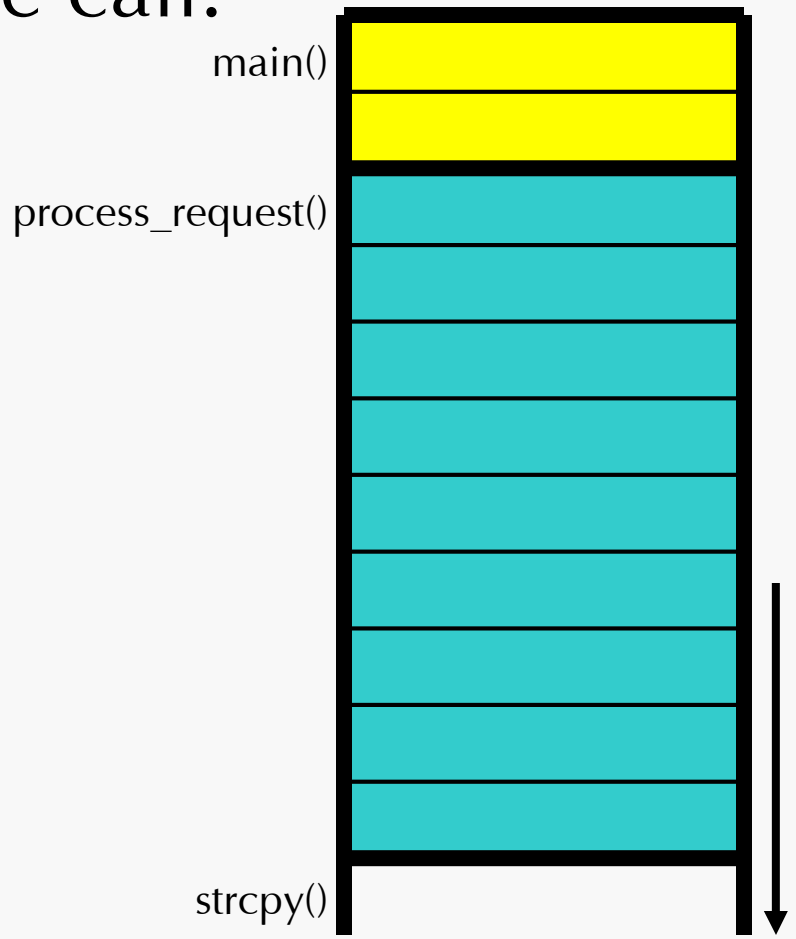
Program Stack

A stack frame per procedure call.

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



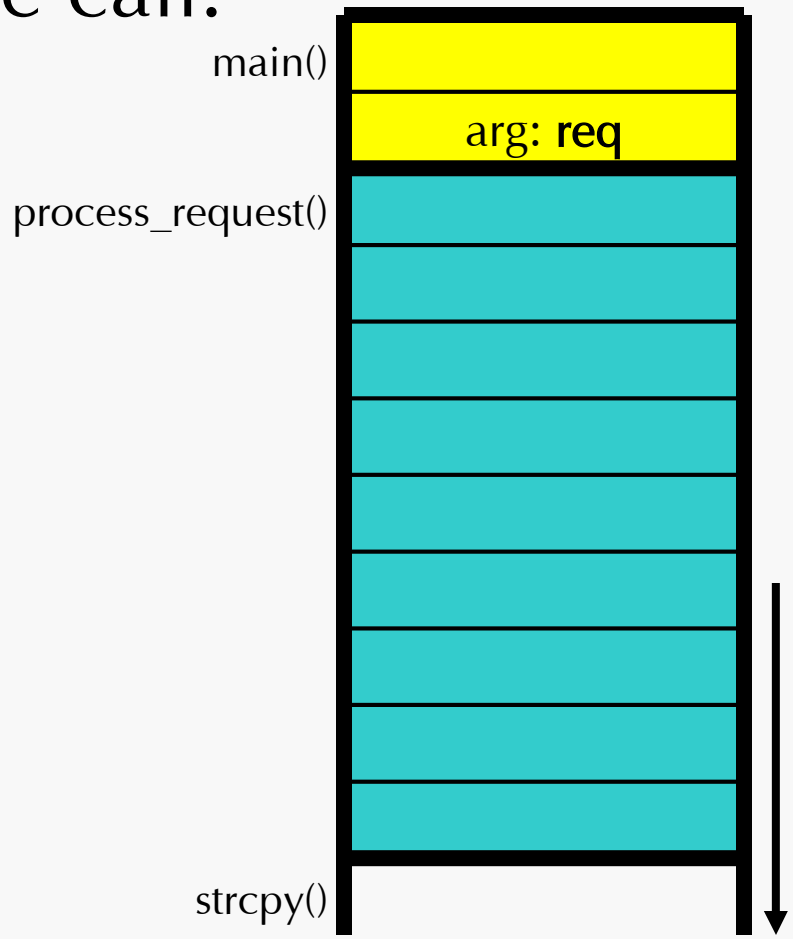
Program Stack

A stack frame per procedure call.

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



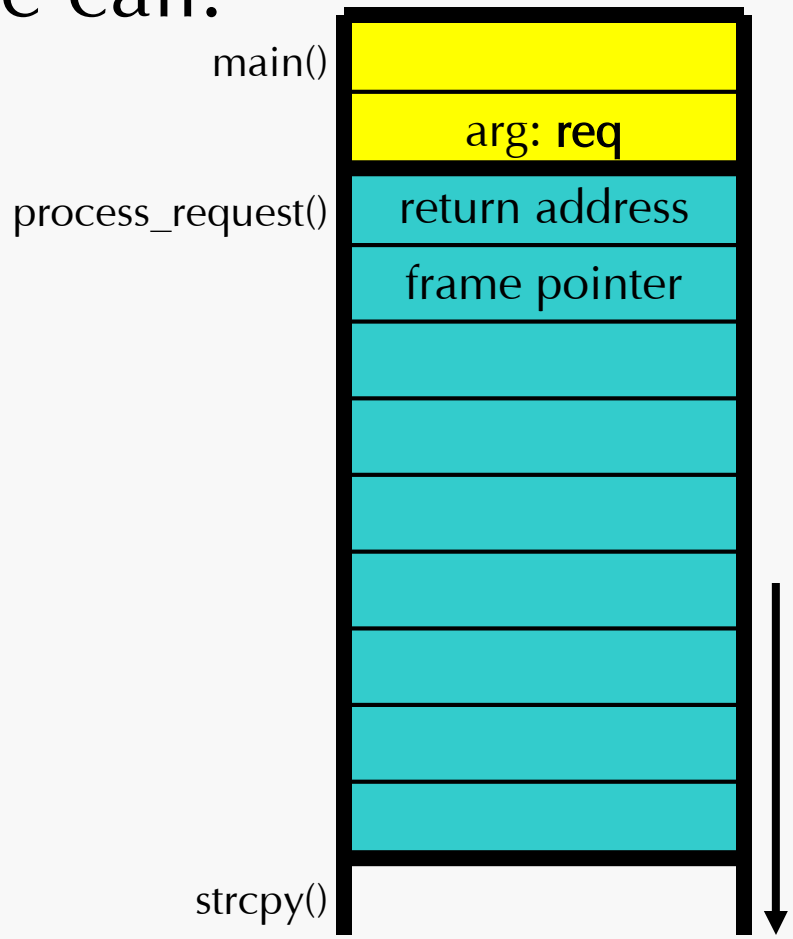
Program Stack

A stack frame per procedure call.

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



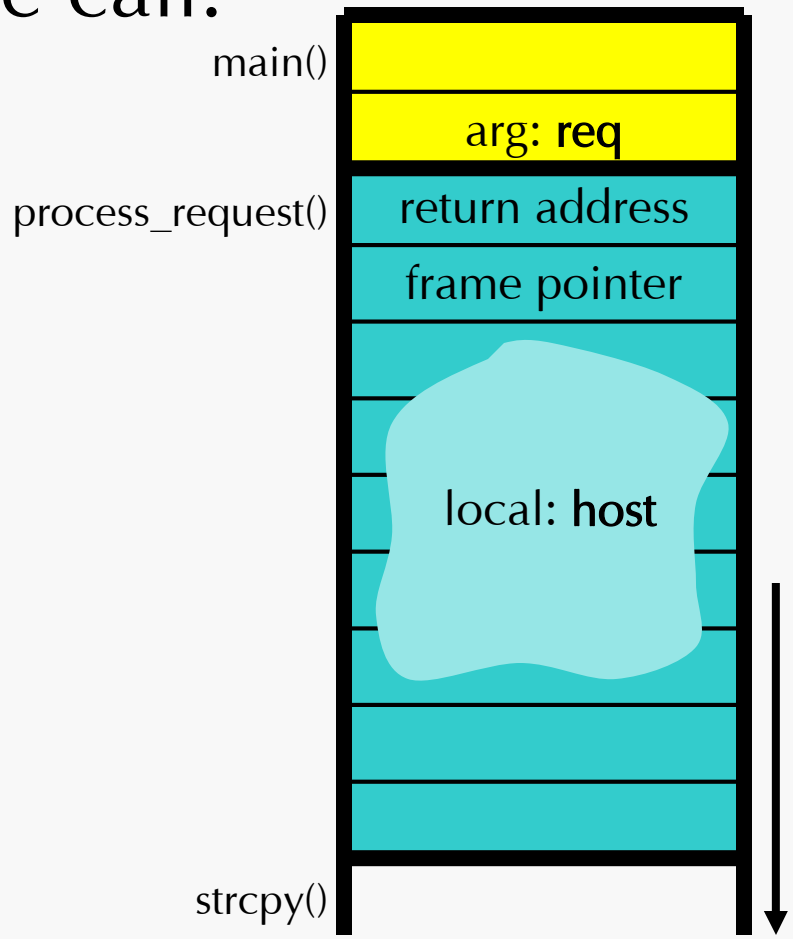
Program Stack

A stack frame per procedure call.

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



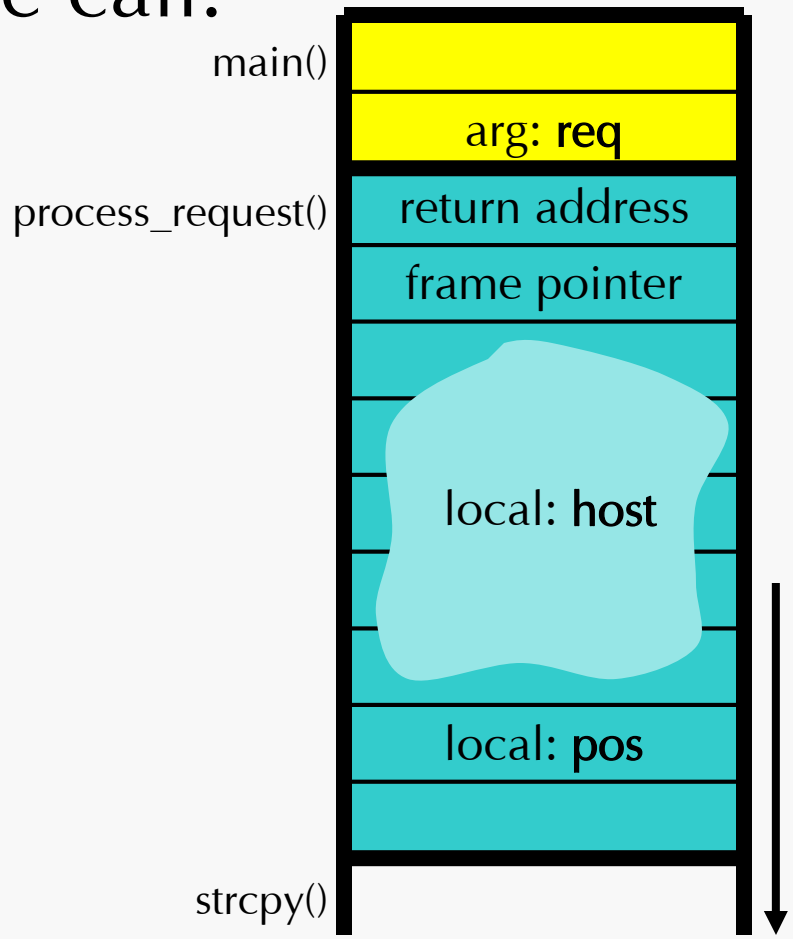
Program Stack

A stack frame per procedure call.

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



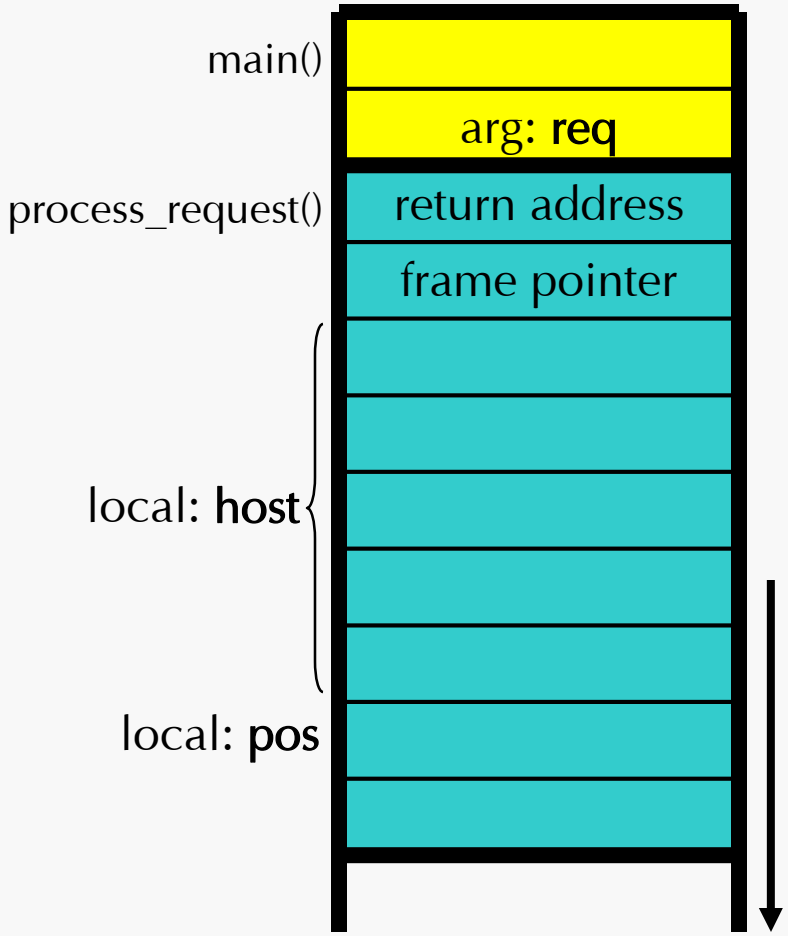
Normal Execution

```
process_request( "\\tux12\usr\foo.txt" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



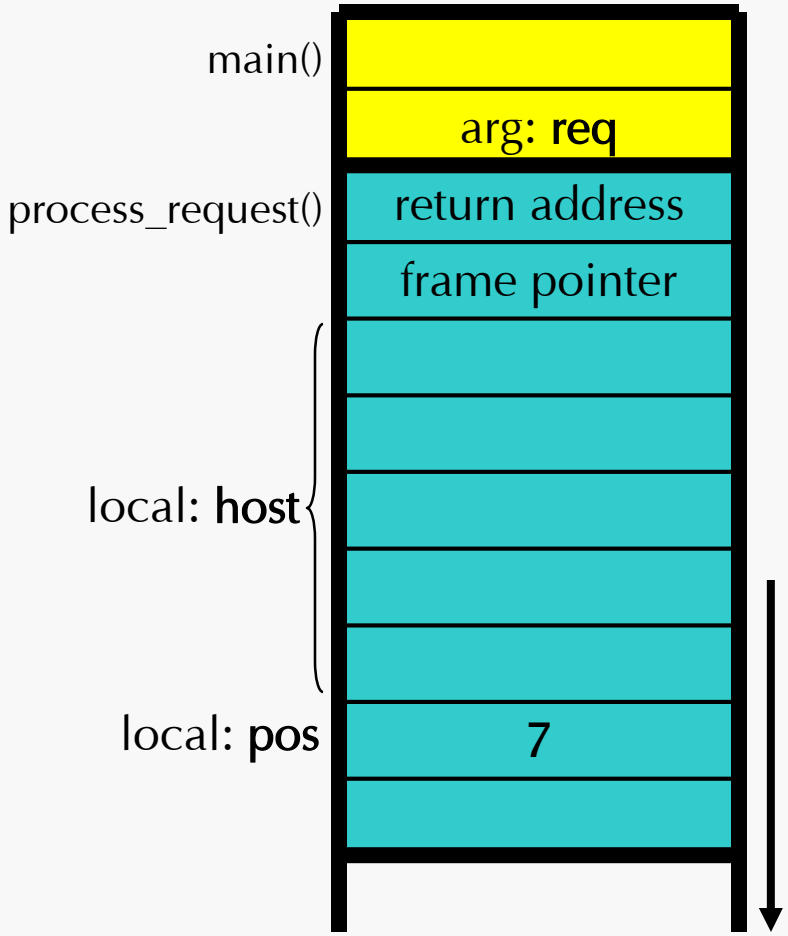
Normal Execution

```
process_request( "\\tux12\usr\foo.txt" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



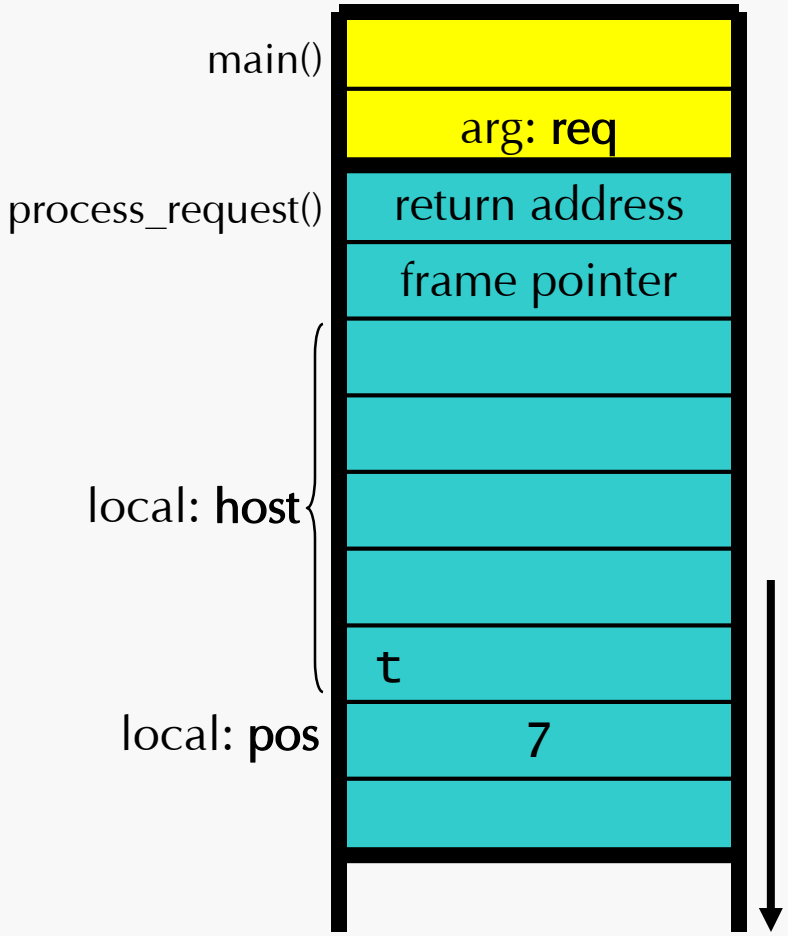
Normal Execution

```
process_request( "\\tux12\usr\foo.txt" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



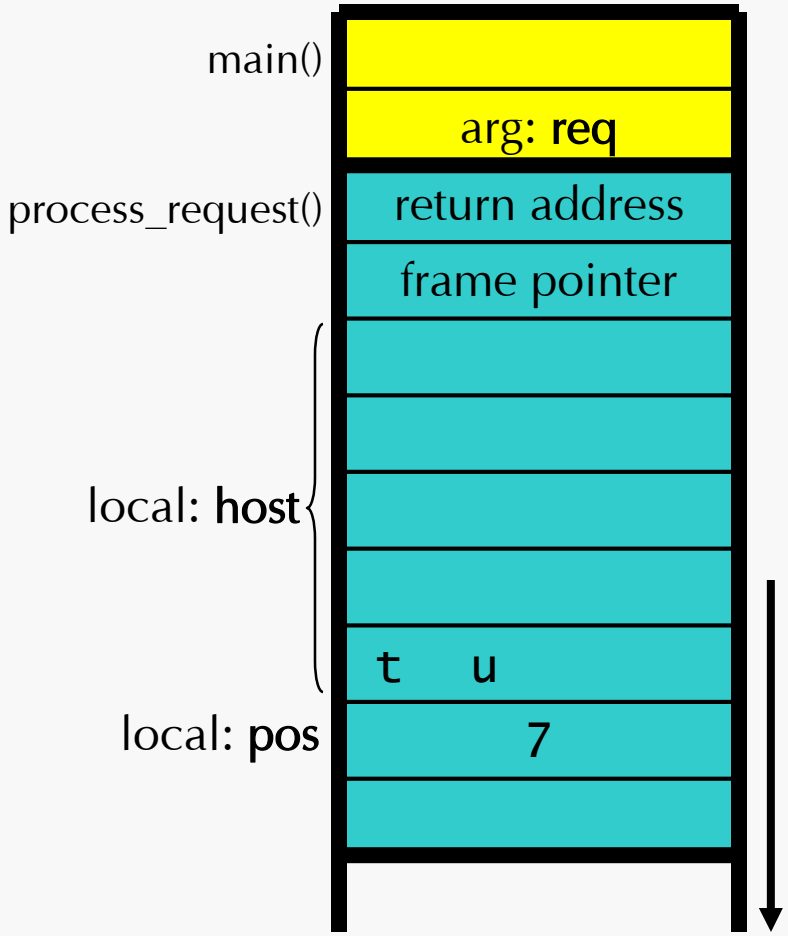
Normal Execution

```
process_request( "\\tux12\usr\foo.txt" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



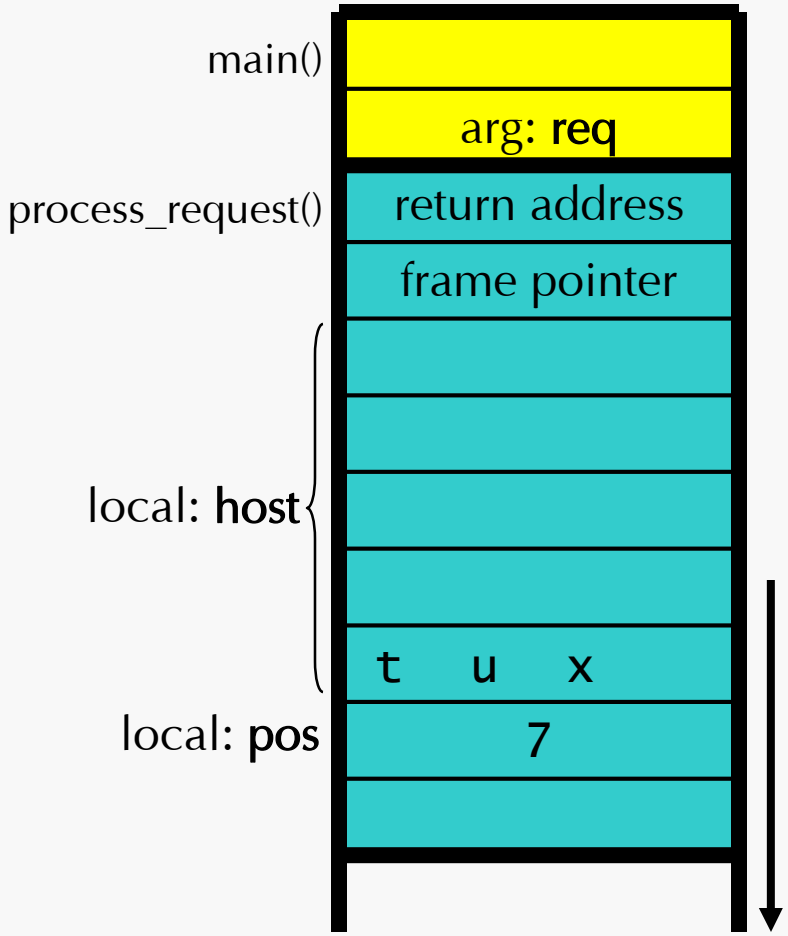
Normal Execution

```
process_request( "\\tux12\usr\foo.txt" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



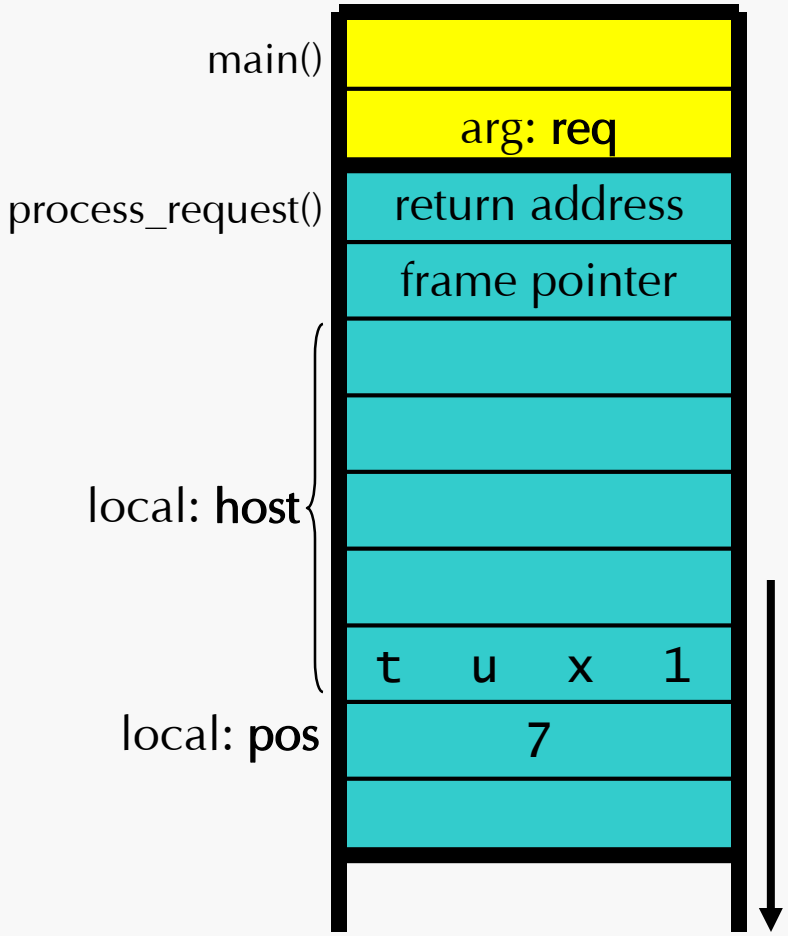
Normal Execution

```
process_request( "\\tux12\usr\foo.txt" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



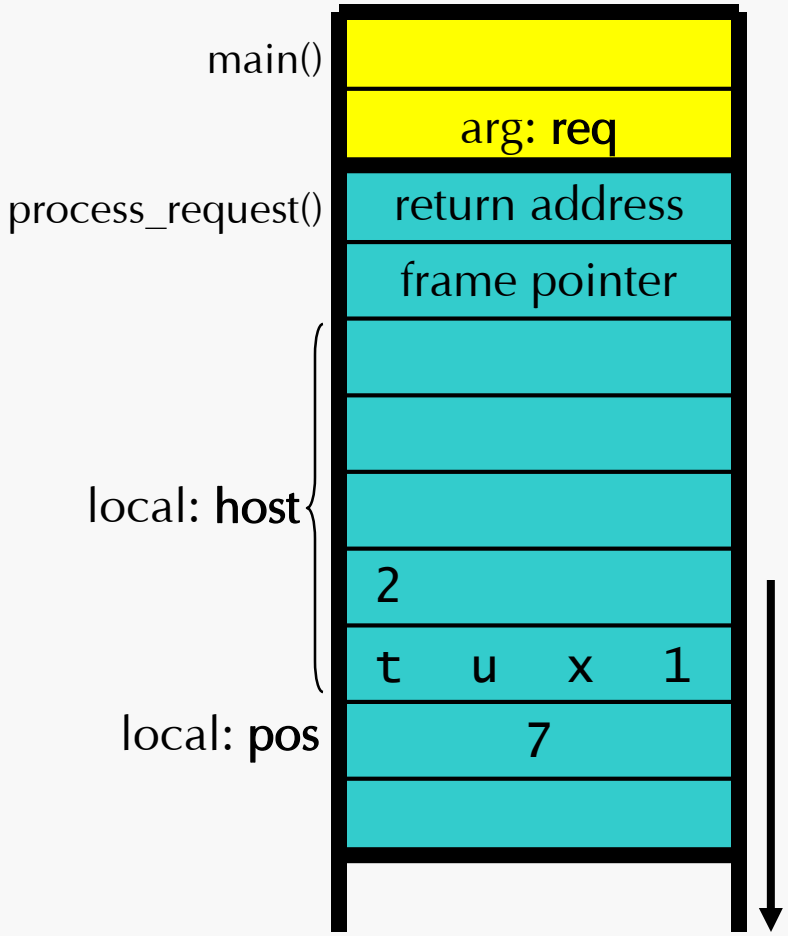
Normal Execution

```
process_request( "\\tux12\usr\foo.txt" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



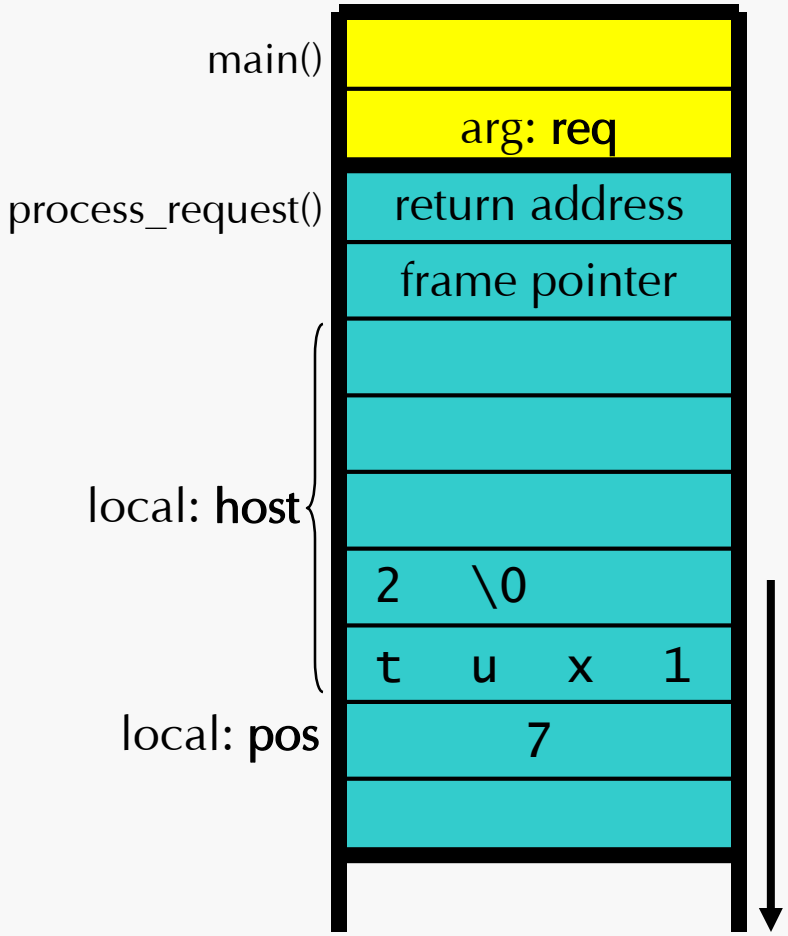
Normal Execution

```
process_request( "\\tux12\usr\foo.txt" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



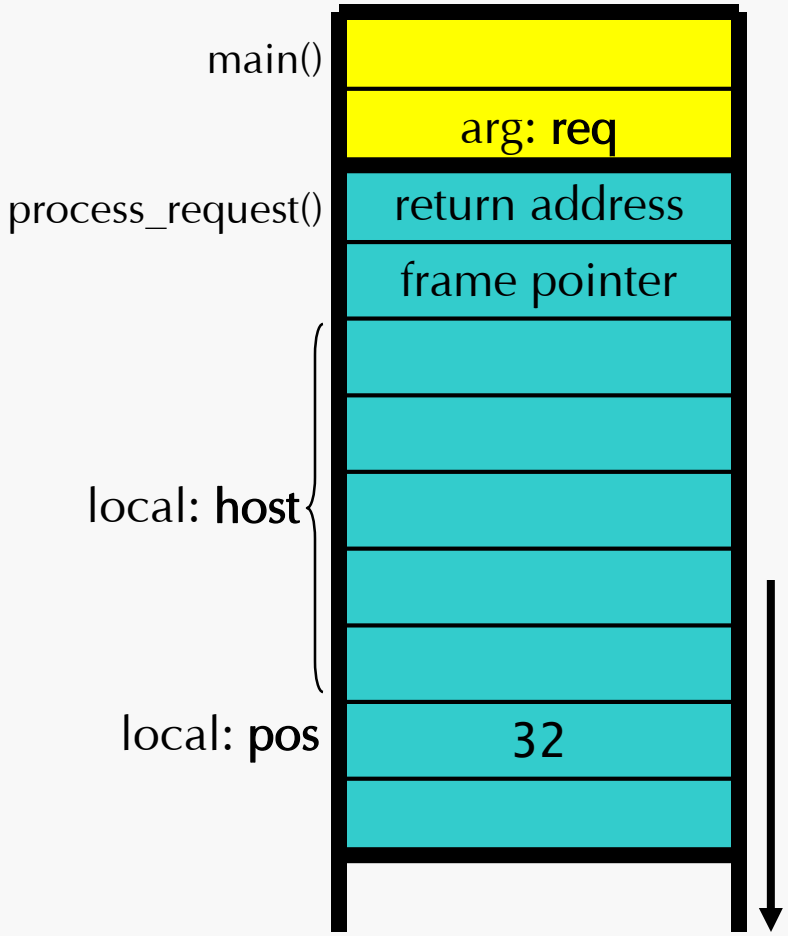
Overflow Execution

```
process_request( "\\aaabbbcccdddeeefffgggghhhiiijjj\bar" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



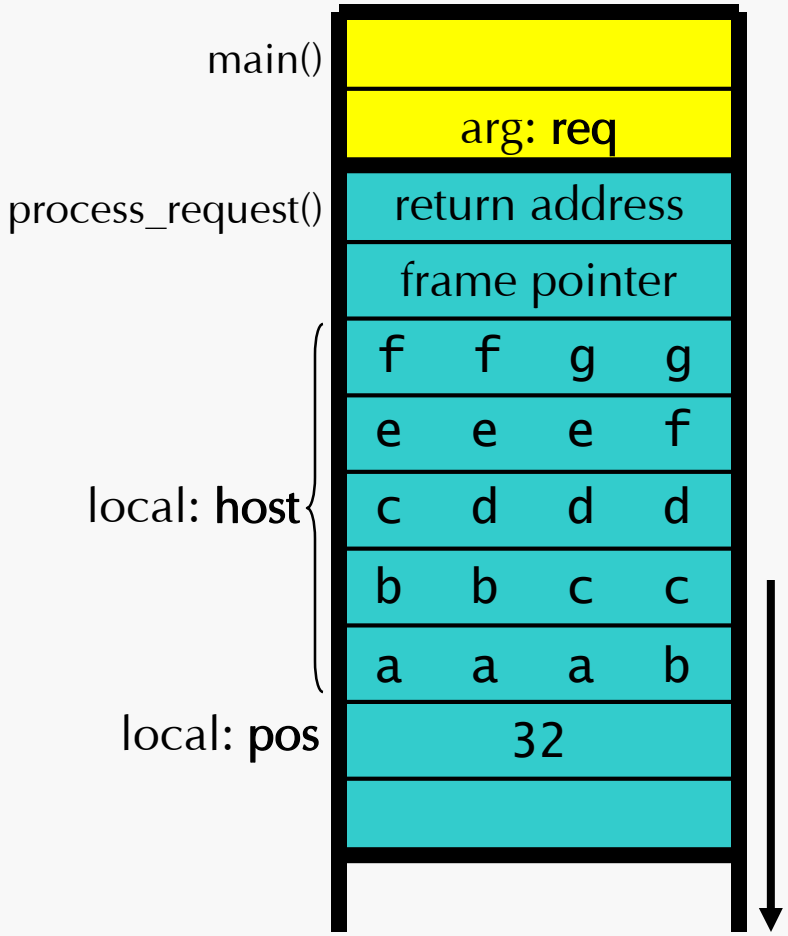
Overflow Execution

```
process_request( "\\aaabbbcccdddeeefffgggghhhiiijjj\bar" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



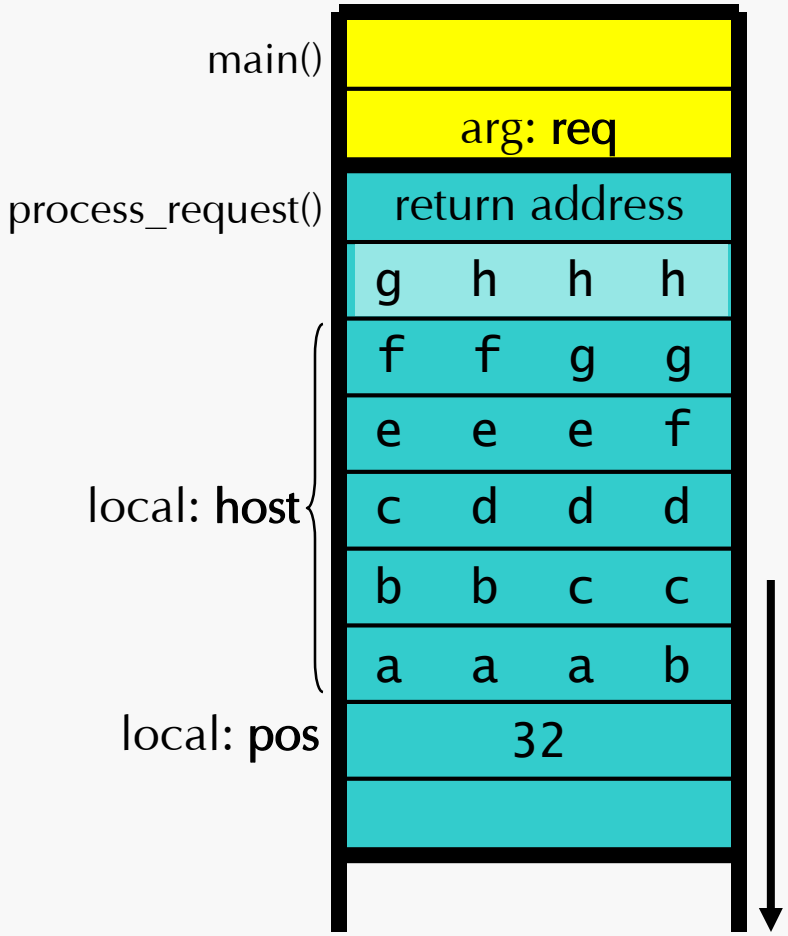
Overflow Execution

```
process_request( "\\aaabbbcccdddeeefffgggghhhiiijjj\bar" );
```

```
void process_request( char * req )
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



Overflow Execution

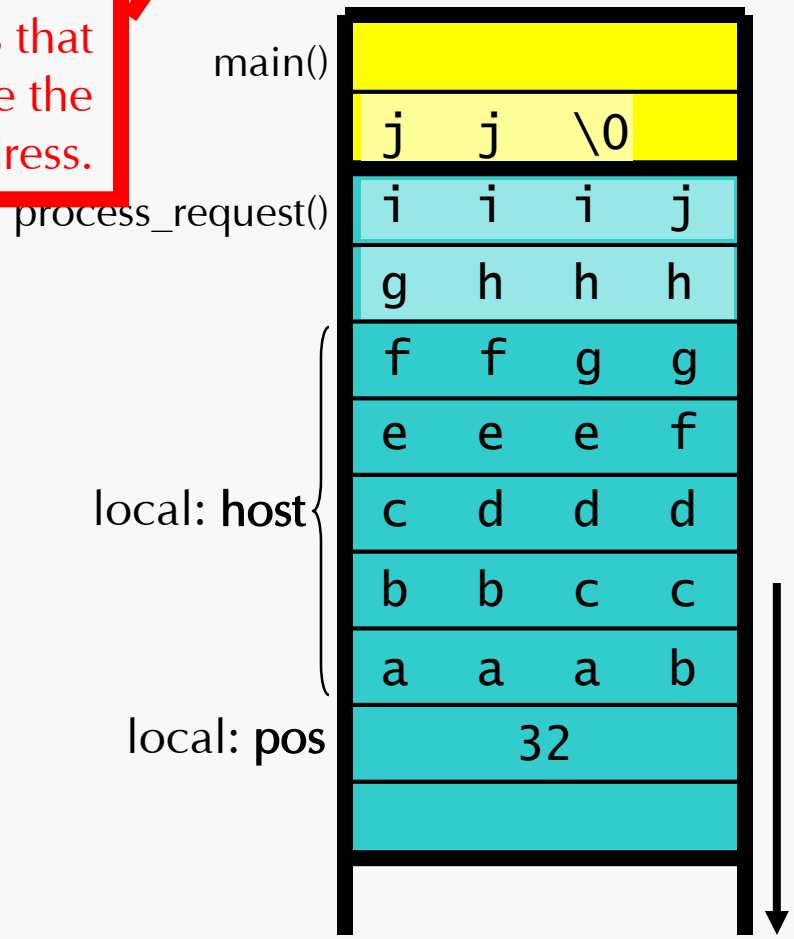
```
process_request( "\\aaabbbcccdddeeefffggghhhiiijjj\bar" );
```

Characters that overwrite the return address.

```
void process_request( char * req
{
   // Get hostname
   char host[ 20 ];
   int pos = find_char( req, '\\', 2 );
   strcpy( host,
           substr( req, 2, pos - 1 ) );

   ...

   return;
}
```



Smashing the Stack

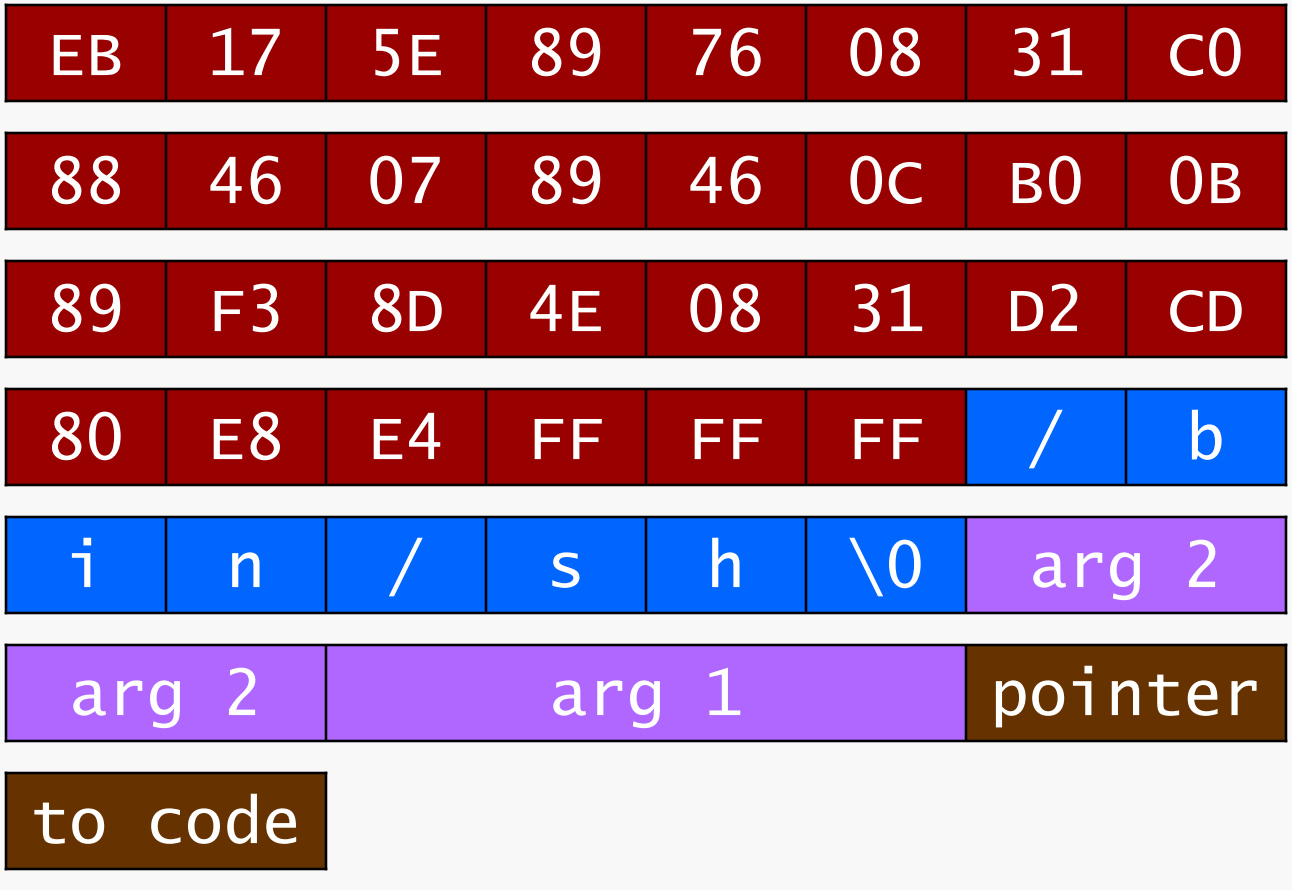
The attacker gets one chance to gain control.

Craft an input string such that:

- The return address is overwritten with a pointer to malicious code.
- The malicious code is placed inside the input string.

Malicious code can create a root shell by executing "`/bin/sh`".

Shell Code



Code for exec("/bin/sh"):

```
mov edx, arg2
mov ecx, arg1
mov ebx, "/bin/sh"
mov eax, 0Bh
int 80h
```

Pointer value for overwriting the return address.

Thicker Armor

- Defense against stack-smashing attacks:
  - Bounds-checking.
  - Protection libraries.
  - Non-executable stack.

  - setuid()/chroot().
  - Avoid running programs as root!
  - Address randomization.

  - Behavioral monitoring.

More Info

"Smashing the Stack for Fun and Profit"
by Aleph One

StackGuard, RAD, PAX, ASLR

CERT

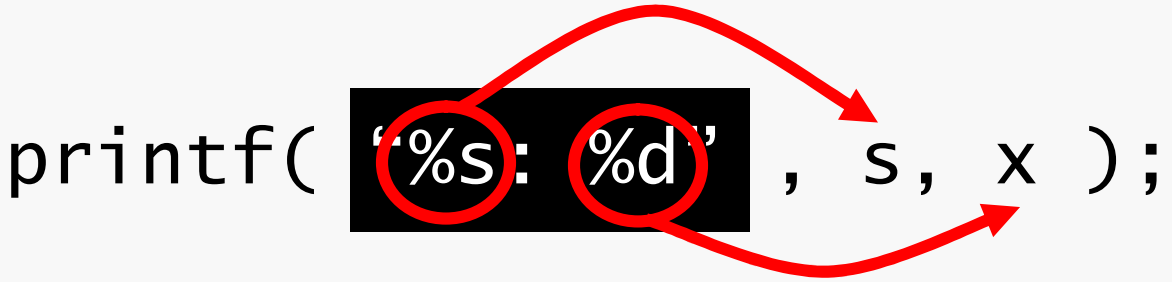
Format String Attacks

- Another way to illegally control program values.
- Uses flaws in the design of `printf()`:

```
printf( “%s: %d” , s, x );
```

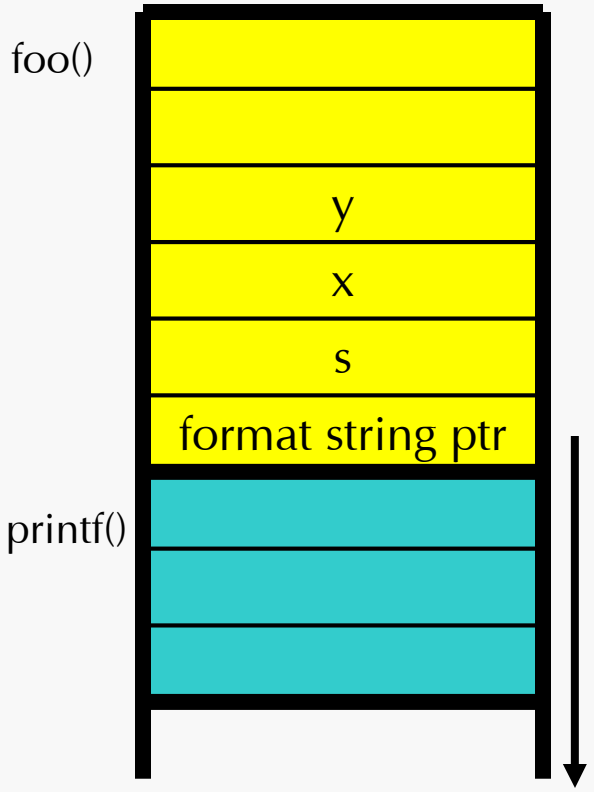
Format String Attacks

- Another way to illegally control program values.
- Uses flaws in the design of `printf()`:



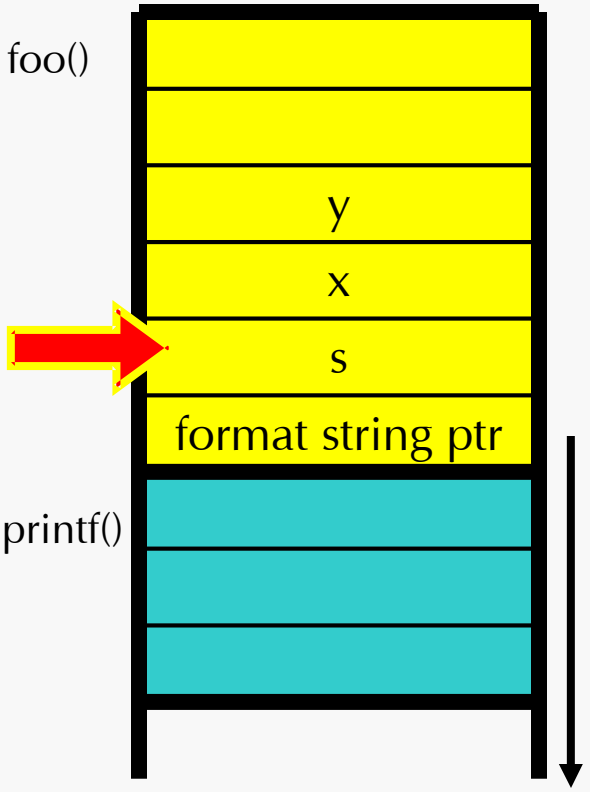
printf() Operation

```
printf( "%s: %d, %x",
        s, x, y );
```



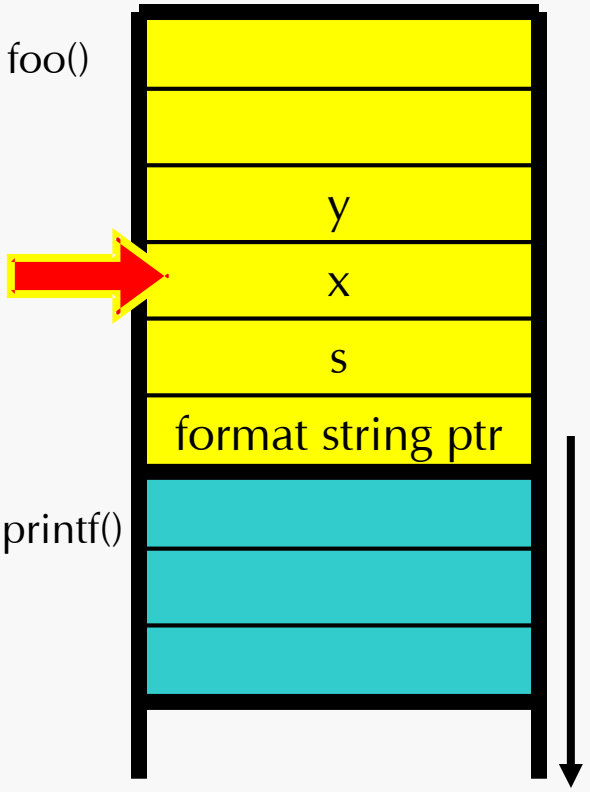
printf() Operation

```
printf( "%s: %d, %x",
        s, x, y );
```



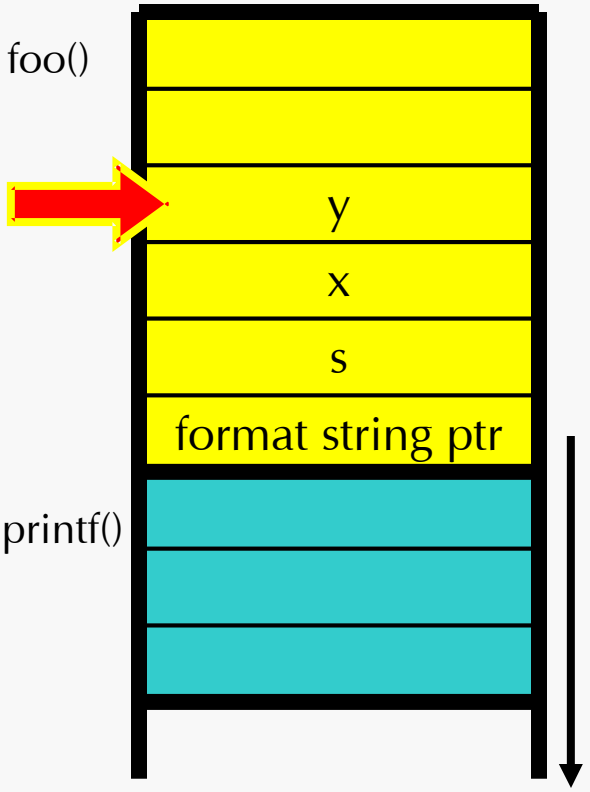
printf() Operation

```
printf( "%s: %d, %x",
        s, x, y );
```



printf() Operation

```
printf( "%s: %d, %x",
        s, x, y );
```



Attack 1: Read Any Value

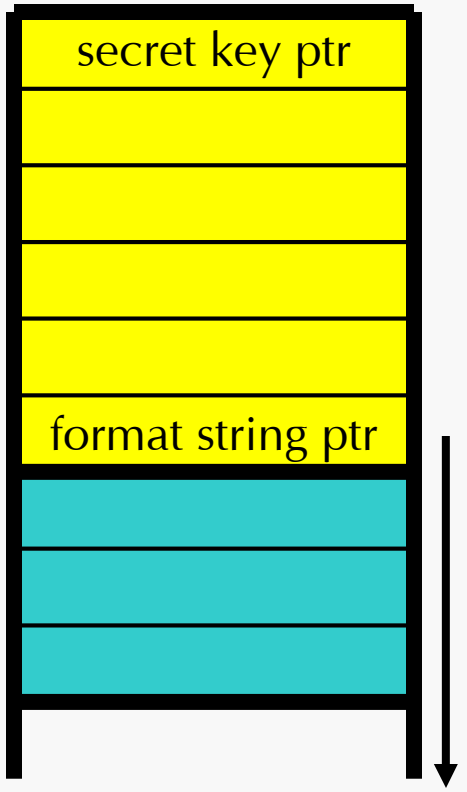
What the code says:

```
printf( str );
```

What the programmer meant:

```
printf( "%s", str );
```

If str = "%x%x%x%x%s"



Attack 1: Read Any Value

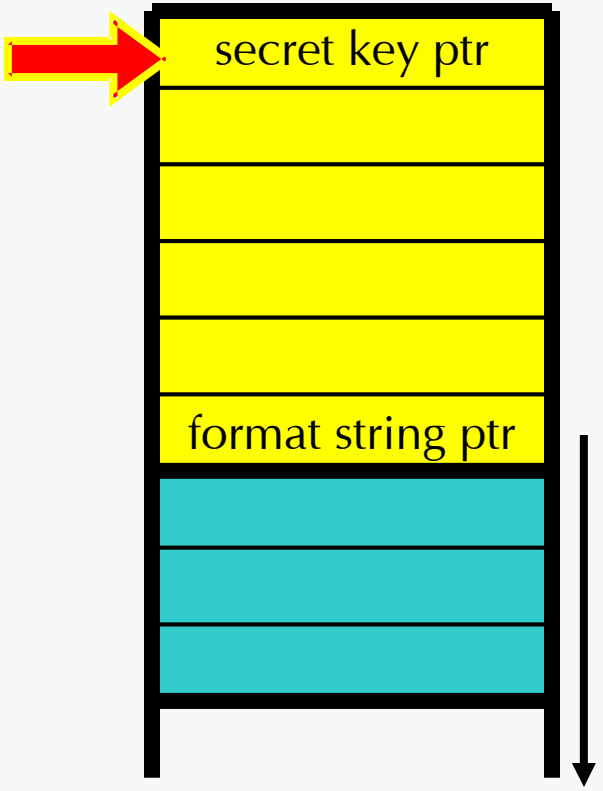
What the code says:

```
printf( str );
```

What the programmer meant:

```
printf( "%s", str );
```

If str = "%x%x%x%x%s"

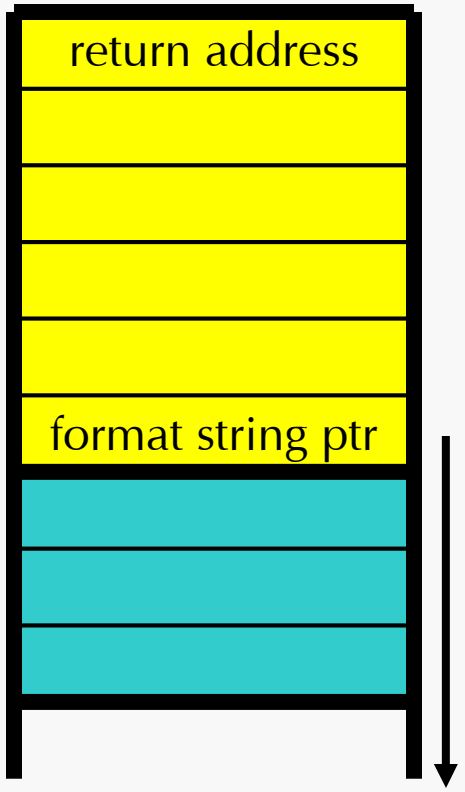


Attack 2: Write to Address

What the code says:

```
printf( str );
```

If str = "%x%x%x%x%n"

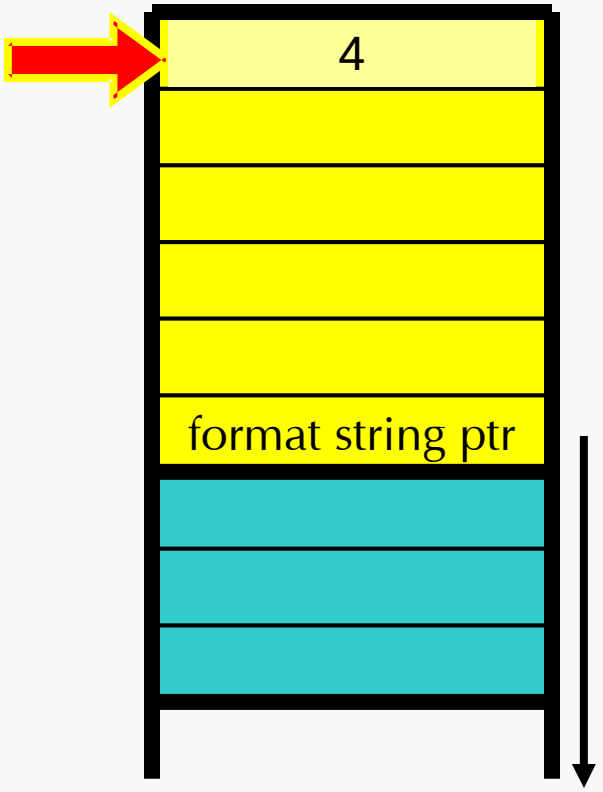


Attack 2: Write to Address

What the code says:

```
printf( str );
```

If str = "%x%x%x%x%n"



Defenses

Never use `printf()` without a format string!

FormatGuard.

Outline

- Attack Vectors:
  - Social Engineering
  - Vulnerability Exploitation
  - Piggybacking
- Payloads
- Spreading Algorithms
- Case Studies

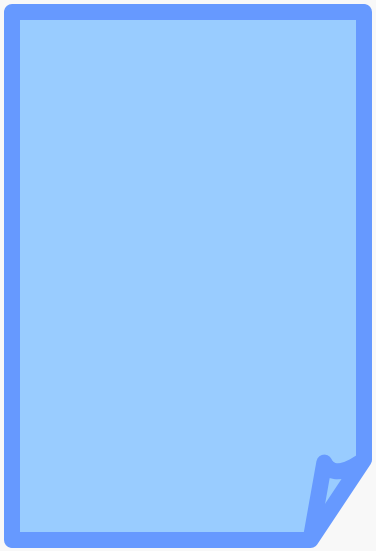
Piggybacking

Malicious code injected into a benign program or data file.

- Host file can be:
  - An executable.
  - A document with some executable content (Word documents with macros, etc.).

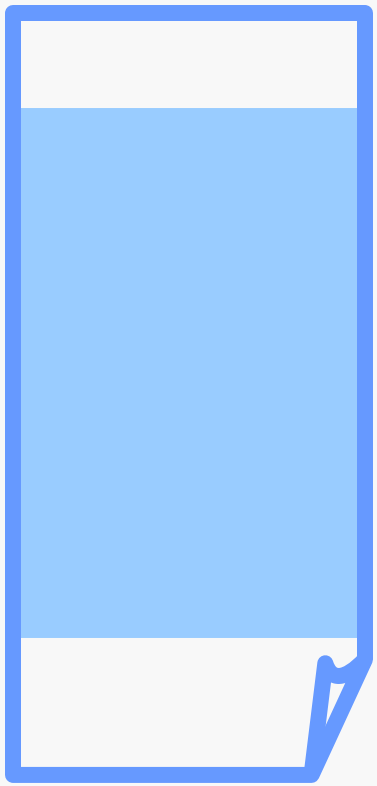
Piggybacking Executables

- Modify program on disk:



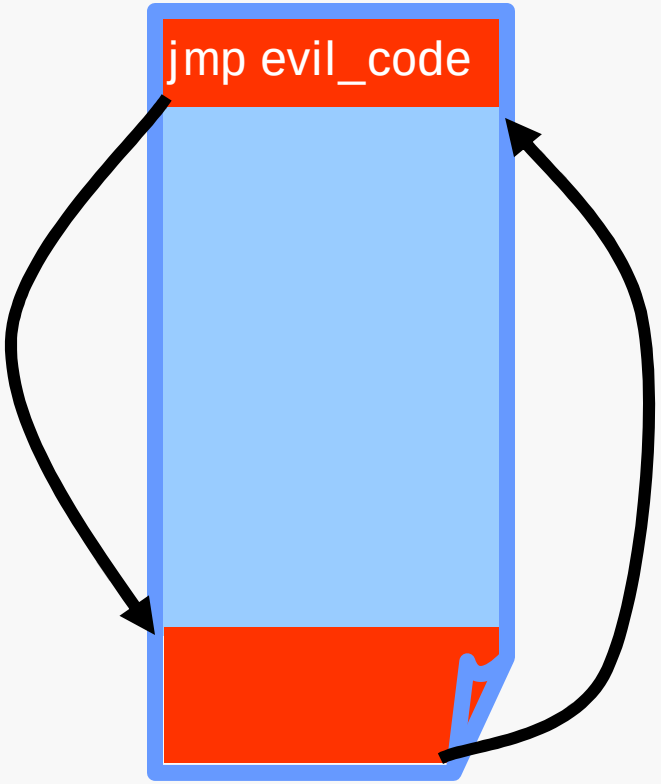
Piggybacking Executables

- Modify program on disk:



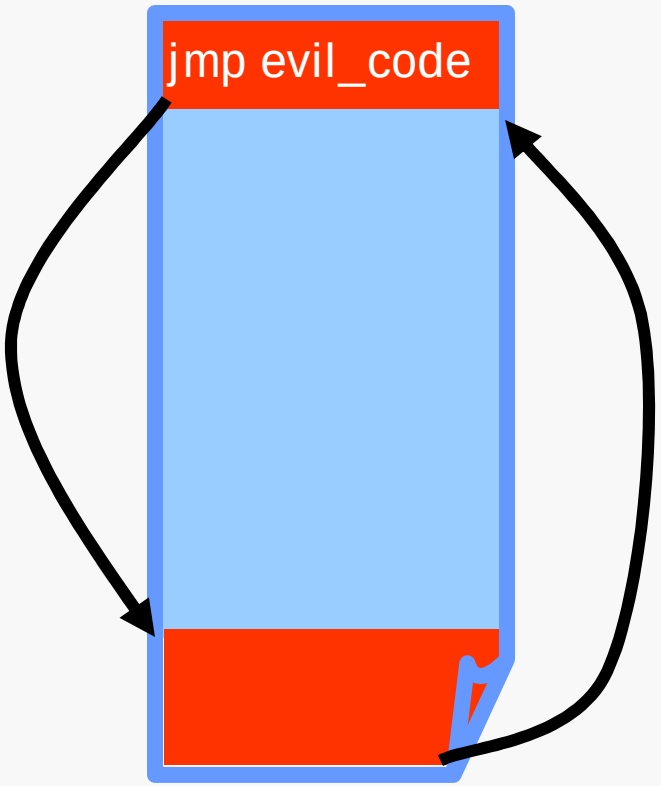
Piggybacking Executables

- Modify program on disk:



Piggybacking Executables

- Modify program on disk:



Variations:

- Jump to malicious code only on certain actions.
- Spread malicious code throughout program.

Piggybacking Documents

- Documents with macros:

  Microsoft Office supports documents with macros scripted in Visual Basic (VBA).

- Macro triggered on:
  - Document open
  - Document close
  - Document save
  - Send document by email

Outline

- Attack Vectors:
  - Social Engineering
  - Vulnerability Exploitation
  - Piggybacking
- Payloads
- Spreading Algorithms
- Case Studies
- Defenses

❷ Payload

Target the interesting data:

- Passwords ⇨ Keylogger
- Financial data ⇨ Screen scraper
- User behavior ⇨ Spyware
- User attention ⇨ Adware

More Payload Ideas

Victim machines are pawns in larger attack:

- Botnets.
- Distributed denial of service (DDoS).
- Spam proxies.
- Anonymous FTP sites.
- IRC servers.

Outline

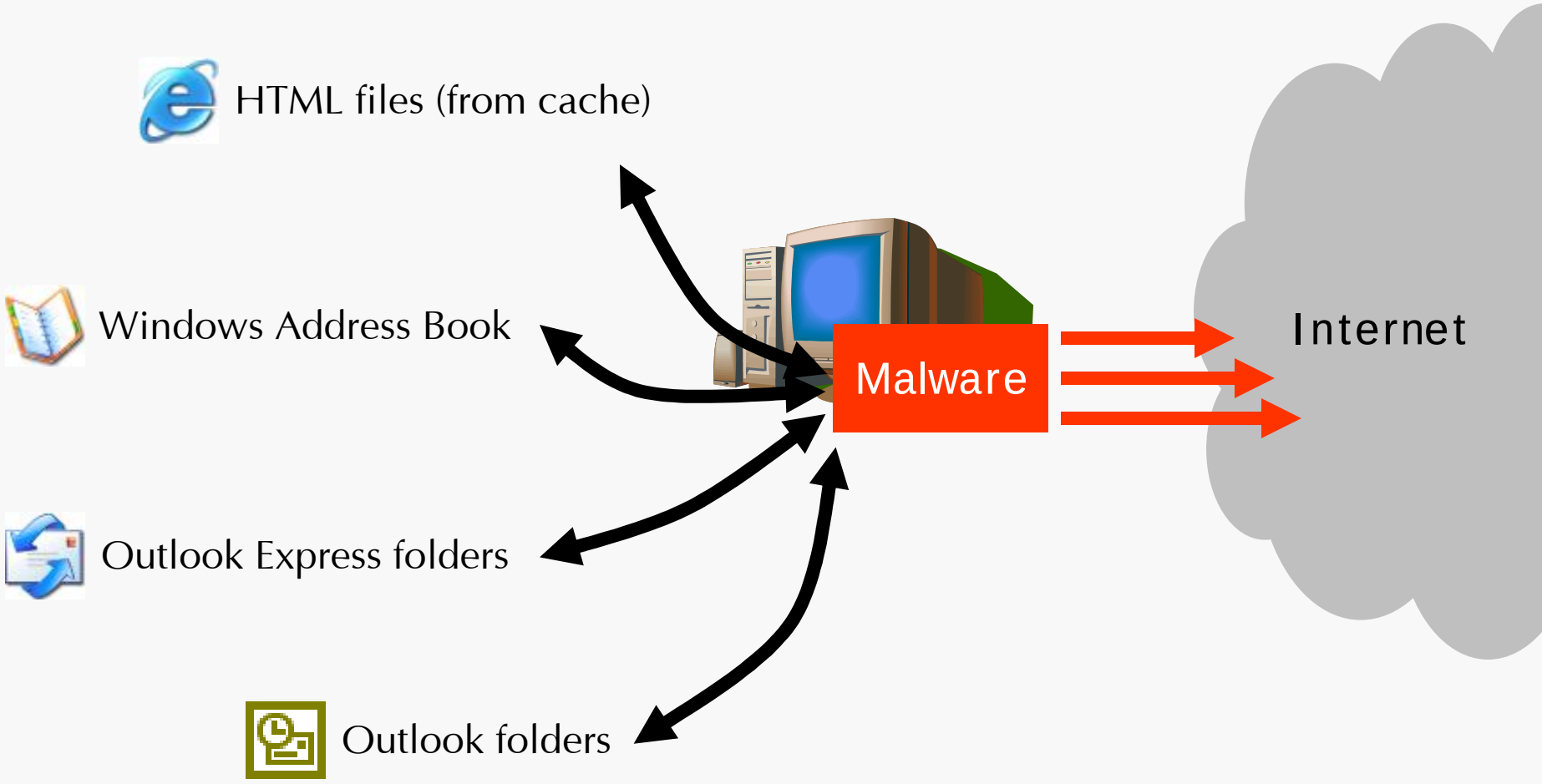
- Attack Vectors:
  - Social Engineering
  - Vulnerability Exploitation
  - Piggybacking
- Payloads
- Spreading Algorithms
- Case Studies
- Defenses

❸ Spreading Methods

Depends on the attack vector:

- Email-based
  - $\Rightarrow$ need email addresses
- Vulnerability-based
  - $\Rightarrow$ need IP addresses of hosts running the vulnerable service
- Piggybacking
  - $\Rightarrow$ need more files to infect

Spreading through Email



Vulnerable Target Discovery

Need to find Internet (IP) addresses.

- Scanning:
  - Random
  - Sequential
  - Bandwidth-limited
- Target list:
  - Pre-generated
  - Externally-generated ⇒ Metaserver worms
  - Internal target list ⇒ Topological worms
- Passive: Contagion worms

Outline

- Attack Vectors:
  - Social Engineering
  - Vulnerability Exploitation
  - Piggybacking
- Payloads
- Spreading Algorithms
- Case Studies

Types of Malicious Code

McGraw and Morrisett "Attacking malicious code: A report to the Infosec Research Council" Sept./Oct. 2000.

- Virus

  Self-replicating, infects programs and documents.

  e.g.: Chernobyl/CIH, Melissa, Elkern

- Worm

  Self-replicating, spreads across a network.

  e.g.: ILoveYou, Code Red, B(e)agle, Witty

Types of Malicious Code

- Trojan
  - Malware hidden inside useful programs

  e.g.: NoUpdate, KillAV, Bookmarker

- Backdoor
  - Tool allowing unauthorized remote access

  e.g.: BackOrifice, SdBot, Subseven

Types of Malicious Code

- Spyware
  - Secretly monitors system activity

  e.g.: ISpynow, KeyLoggerPro, Look2me

- Adware
  - Monitors user activity for advertising purposes

  e.g.: WildTangent, Gator, BargainBuddy

Outline

- Attack Vectors:
  - Social Engineering
  - Vulnerability Exploitation
  - Piggybacking
- Payloads
- Spreading Algorithms
- Case Studies: Sobig

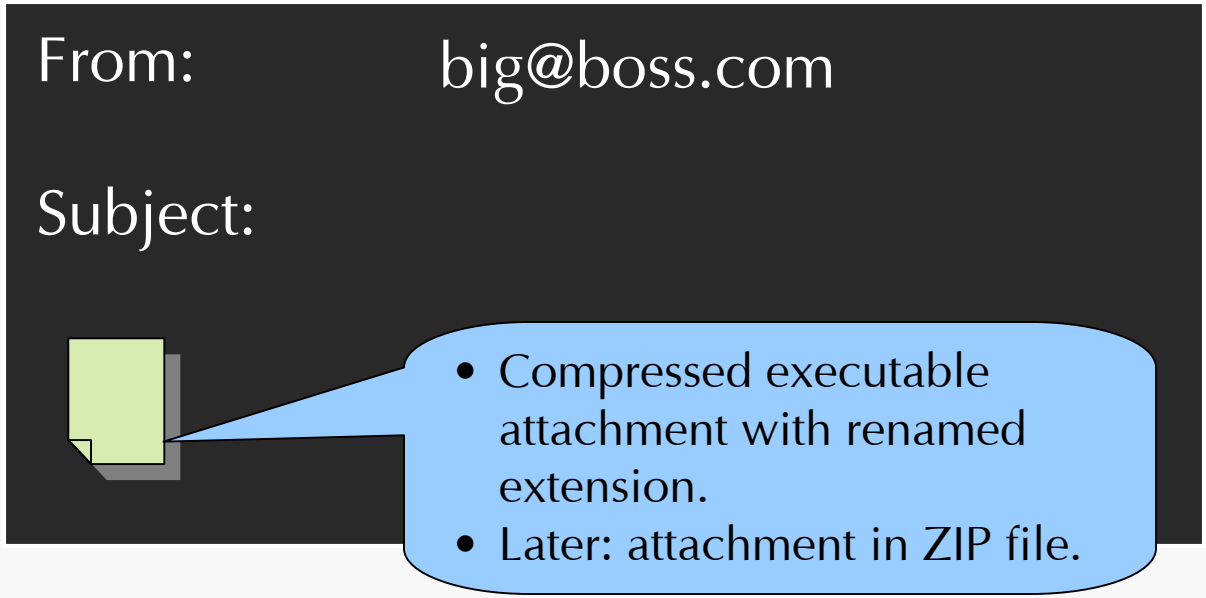
The Sobig Worm

- Mass-mailing, network-aware worm
- Multi-stage update capabilities

	Launch	*Deactivation*
Sobig.A	9 Jan. 2003	-
Sobig.B	18 May 2003	31 May 2003
Sobig.C	31 May 2003	8 June 2003
Sobig.D	18 June 2003	2 July 2003
Sobig.E	25 June 2003	14 July 2003
Sobig.F	18 Aug 2003	10 Sept 2003

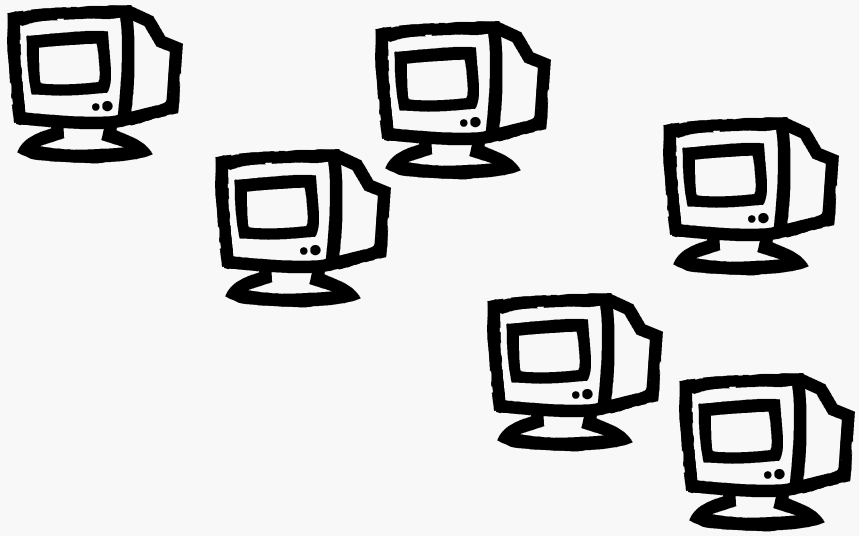
Sobig: Attack Vector

- E-mail

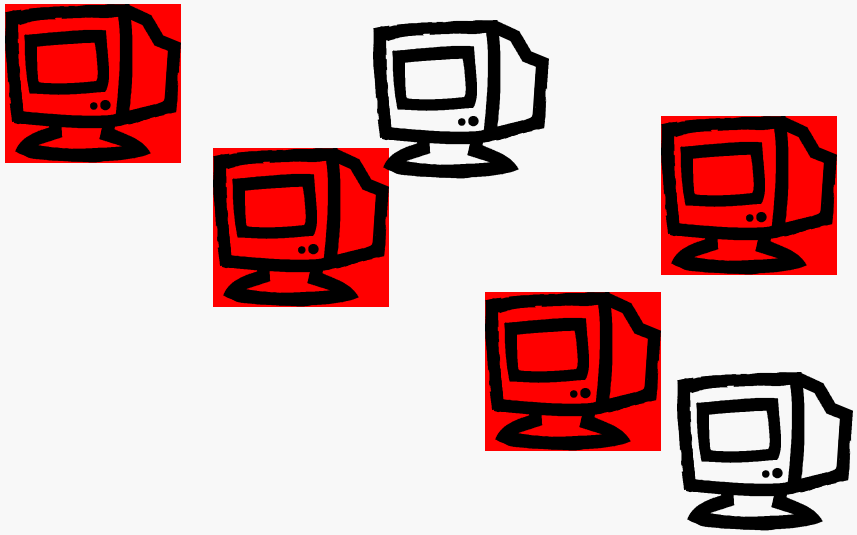


- Network shares

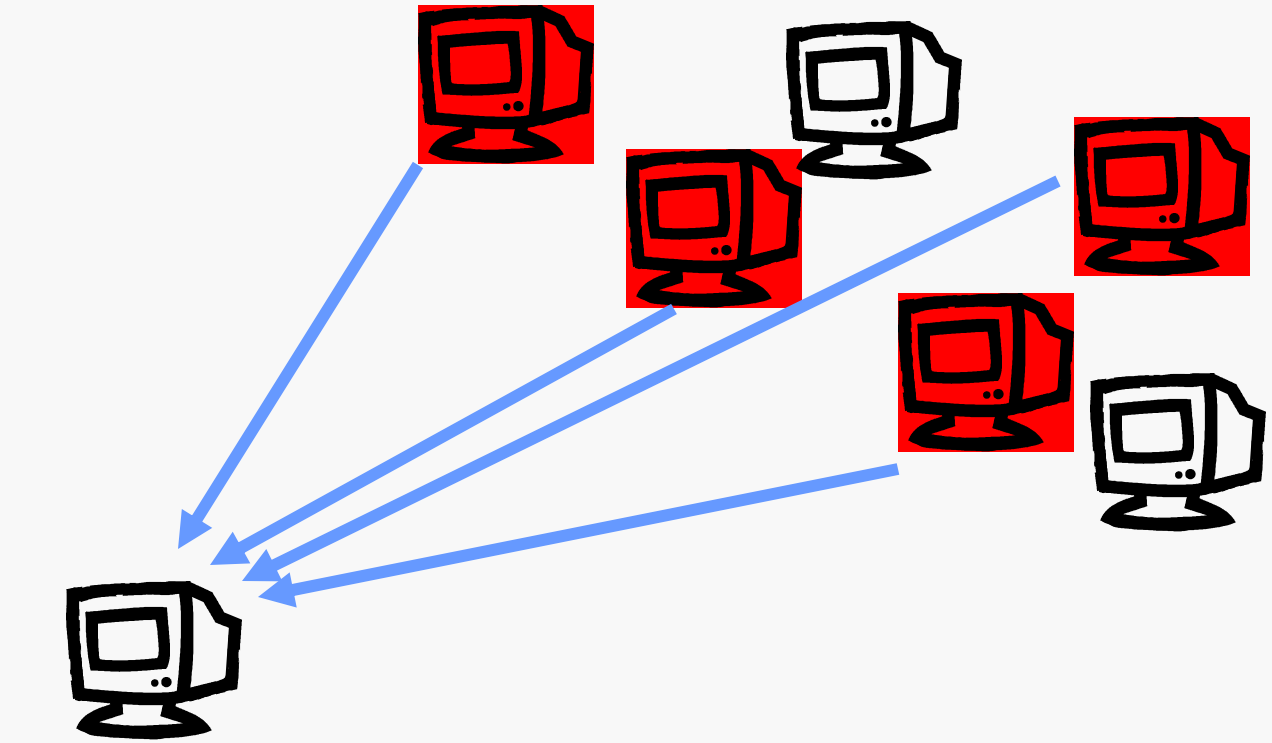
Sobig: Payload



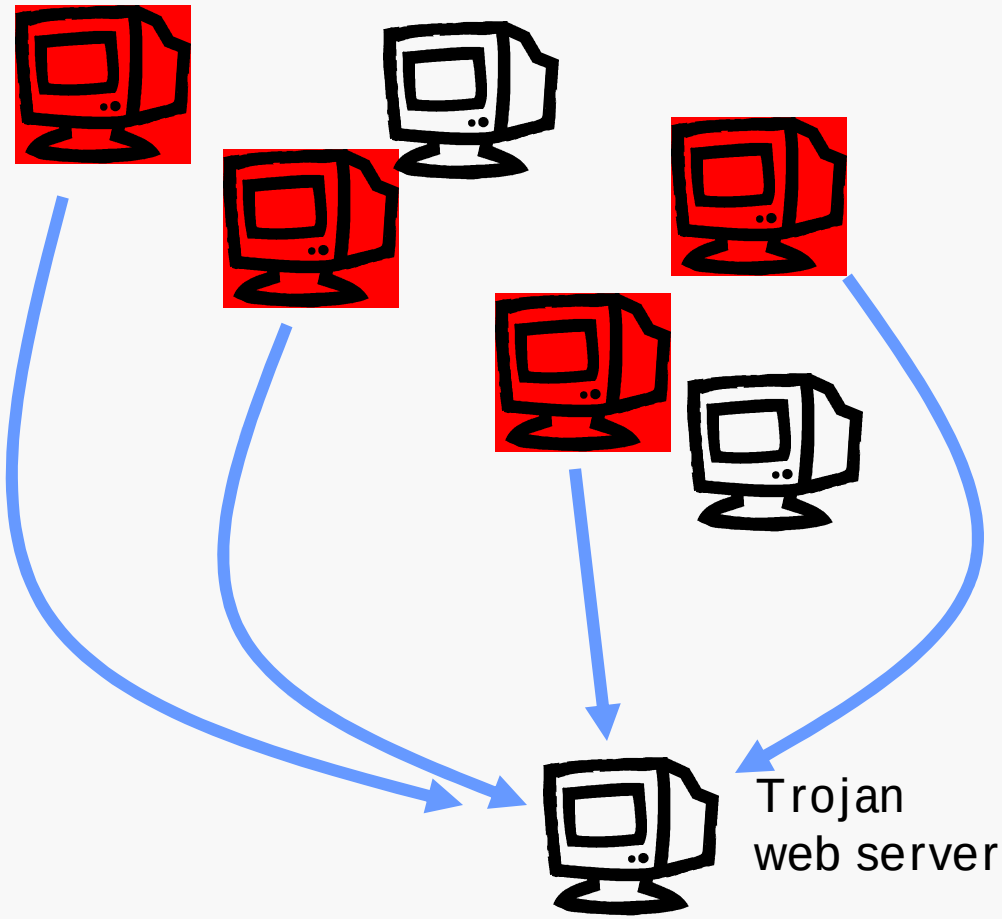
Sobig: Payload



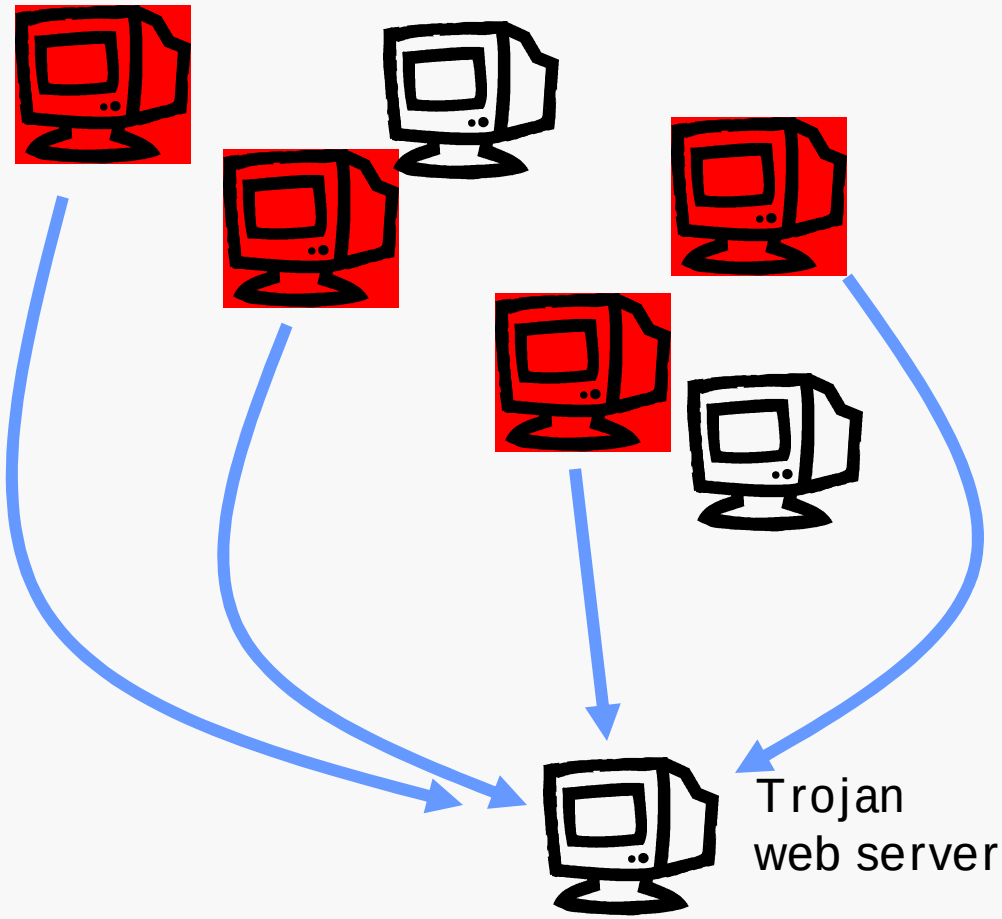
Sobig: Payload



Sobig: Payload

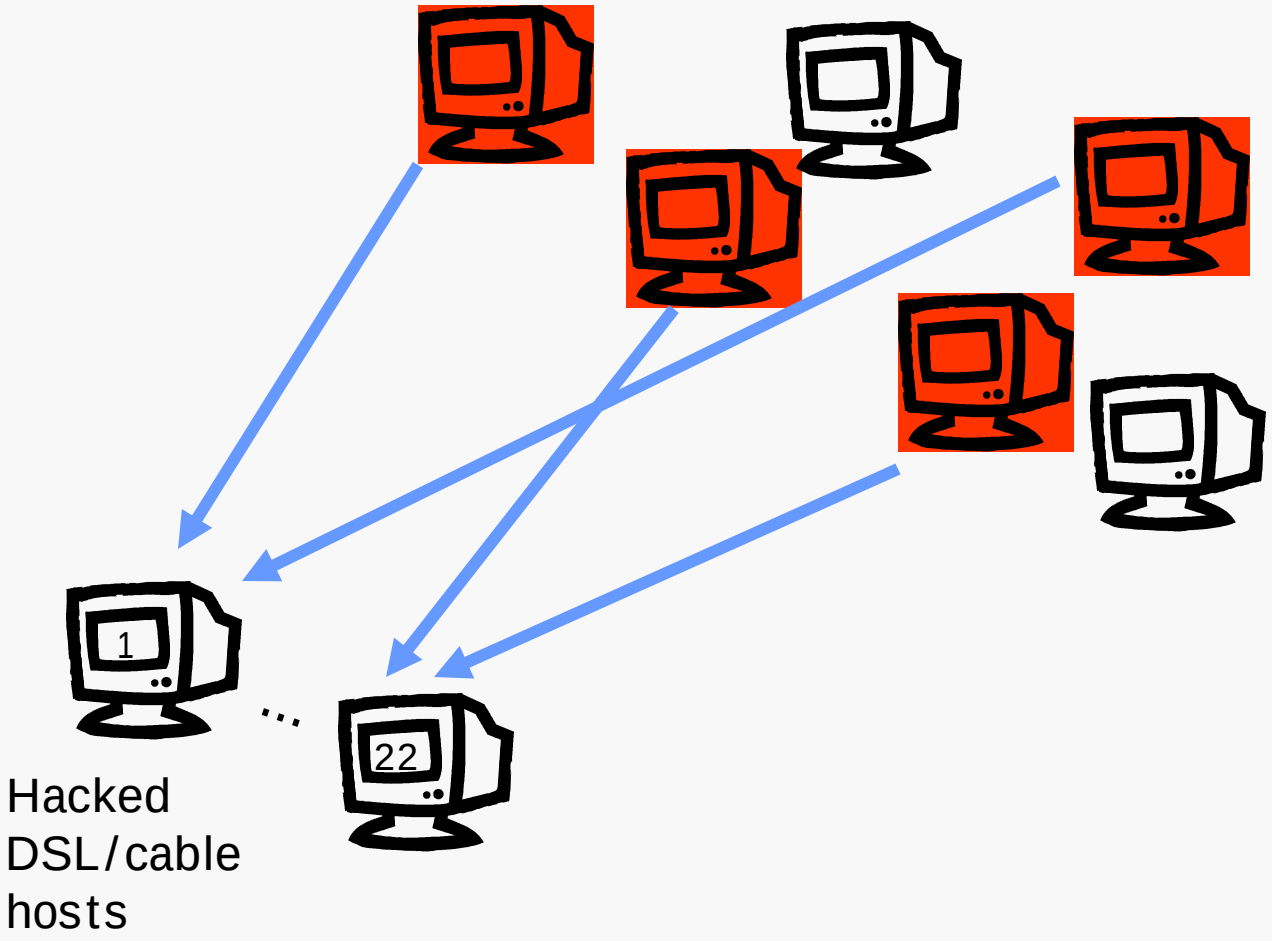


Sobig: Payload

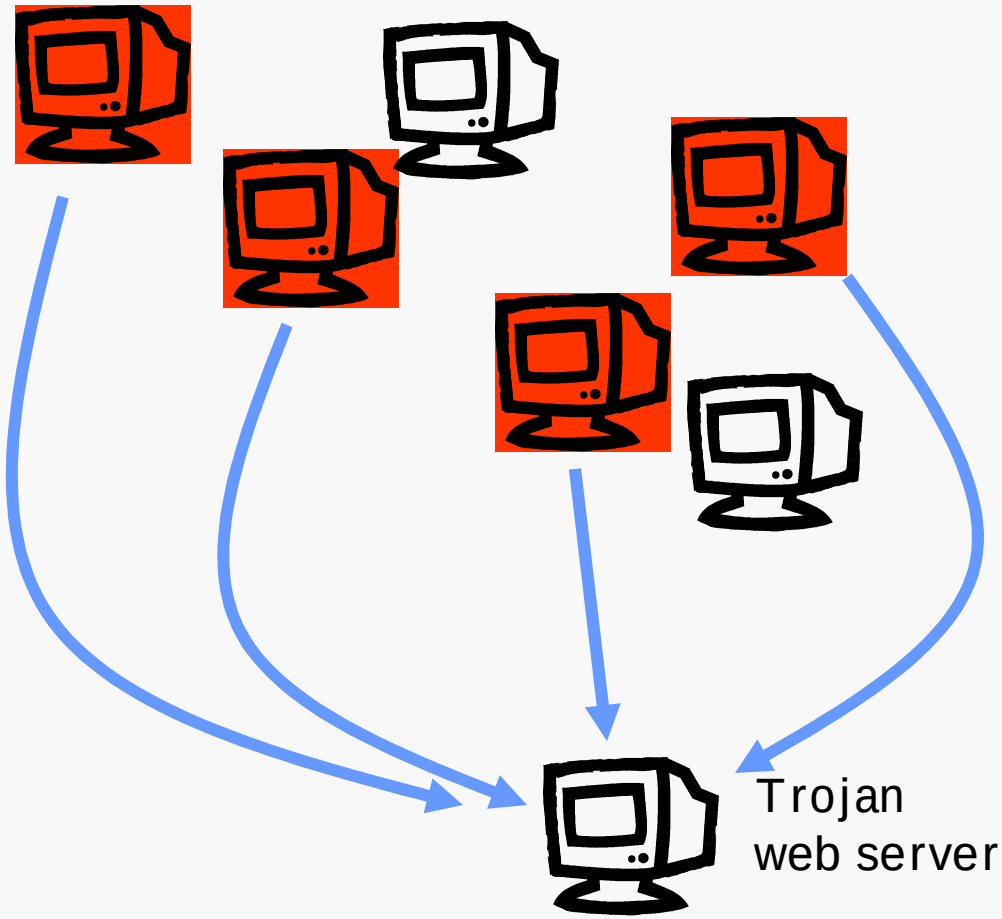


- 1st stage: Backdoor (Lala) & keylogger
- 2nd stage: Proxy (WinGate)

Sobig: Payload



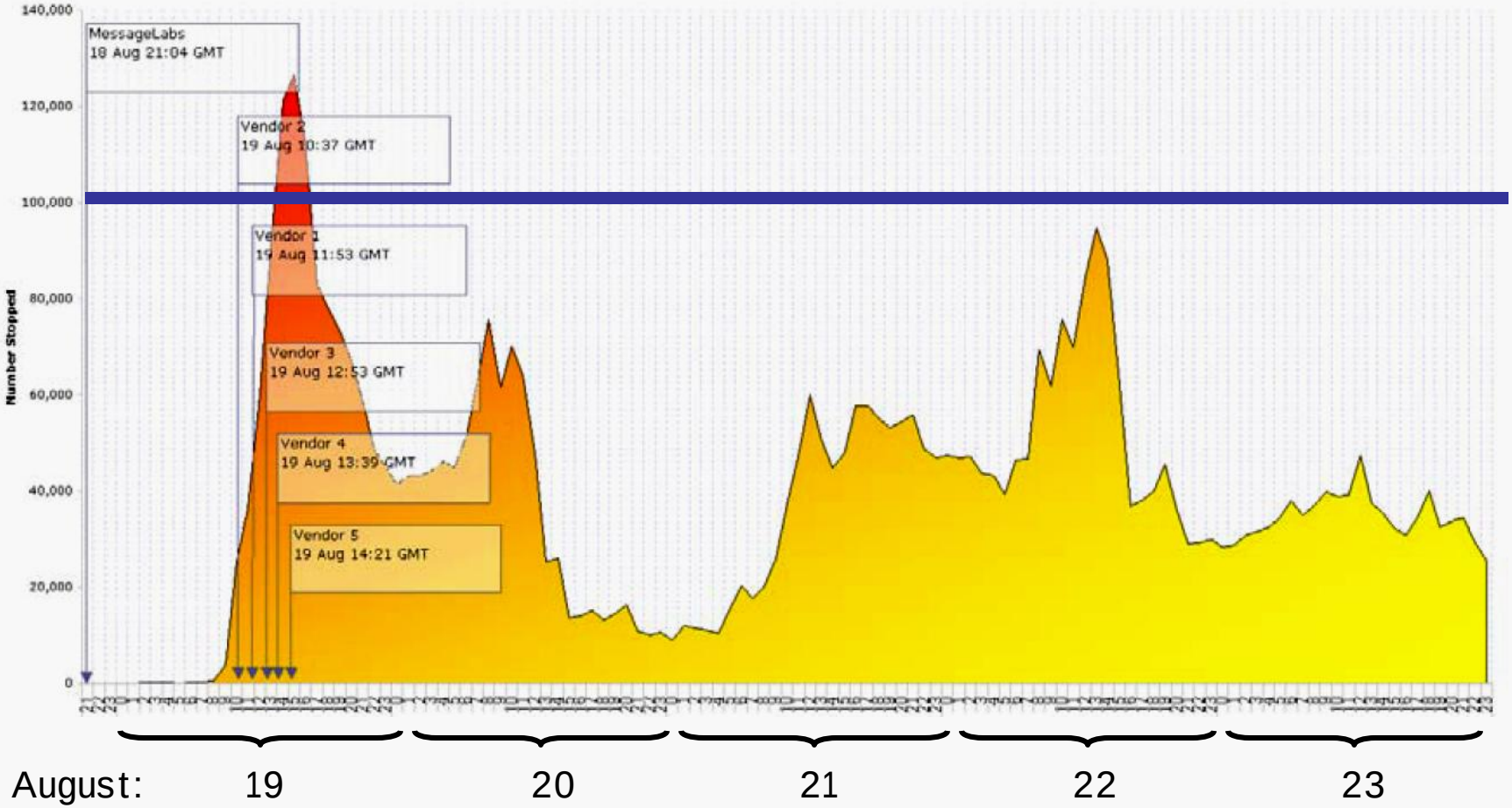
Sobig: Payload



Sobig: Spreading Algorithm

- E-mail addresses extracted from files on disk.
- Network shares automatically discovered.

Sobig.F in Numbers



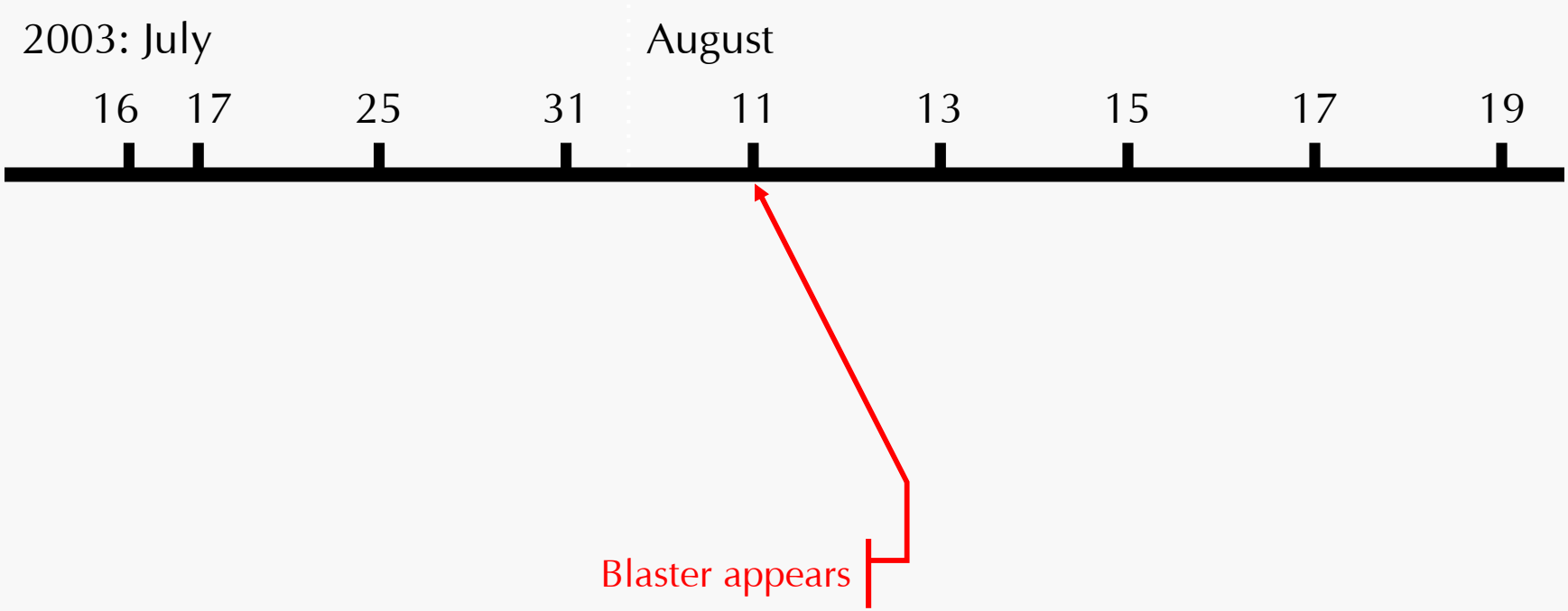
Courtesy of MessageLabs.com

Outline

- Attack Vectors:
  - Social Engineering
  - Vulnerability Exploitation
  - Piggybacking
- Payloads
- Spreading Algorithms
- Case Studies: Sobig, Blaster

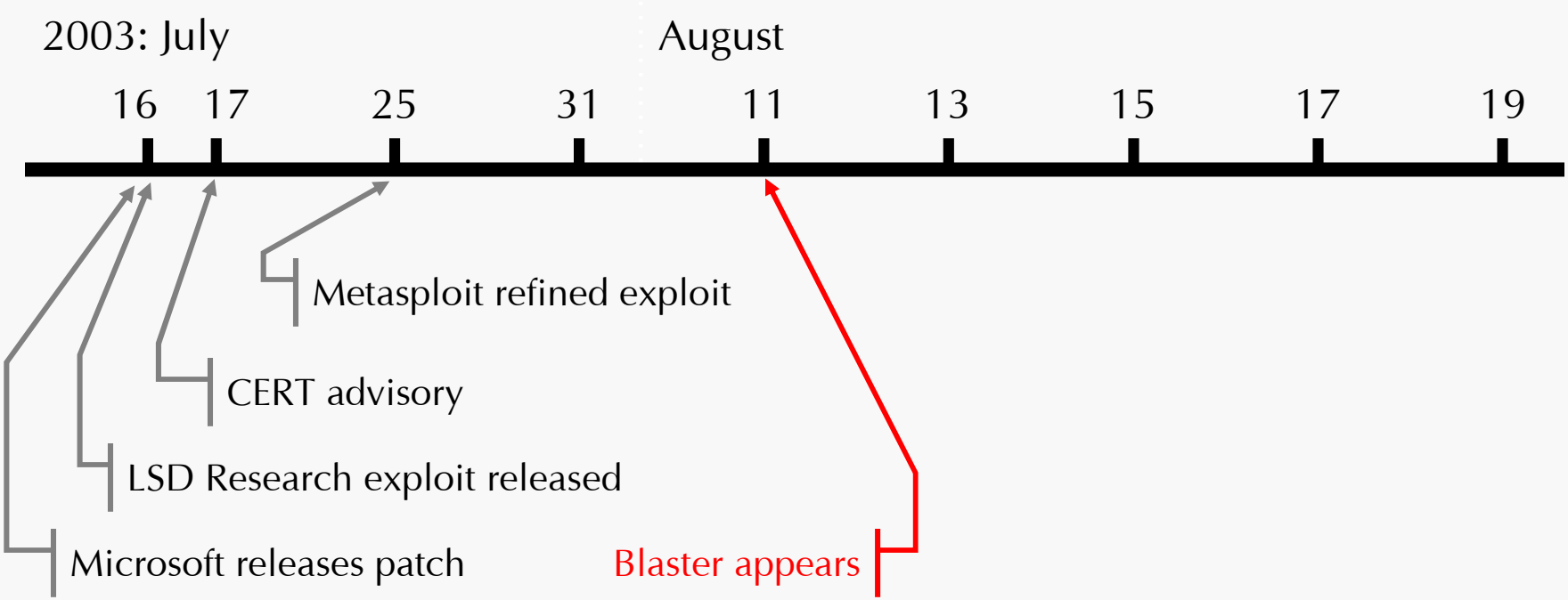
The Blaster Worm

- Multi-stage worm exploiting Windows vulnerability



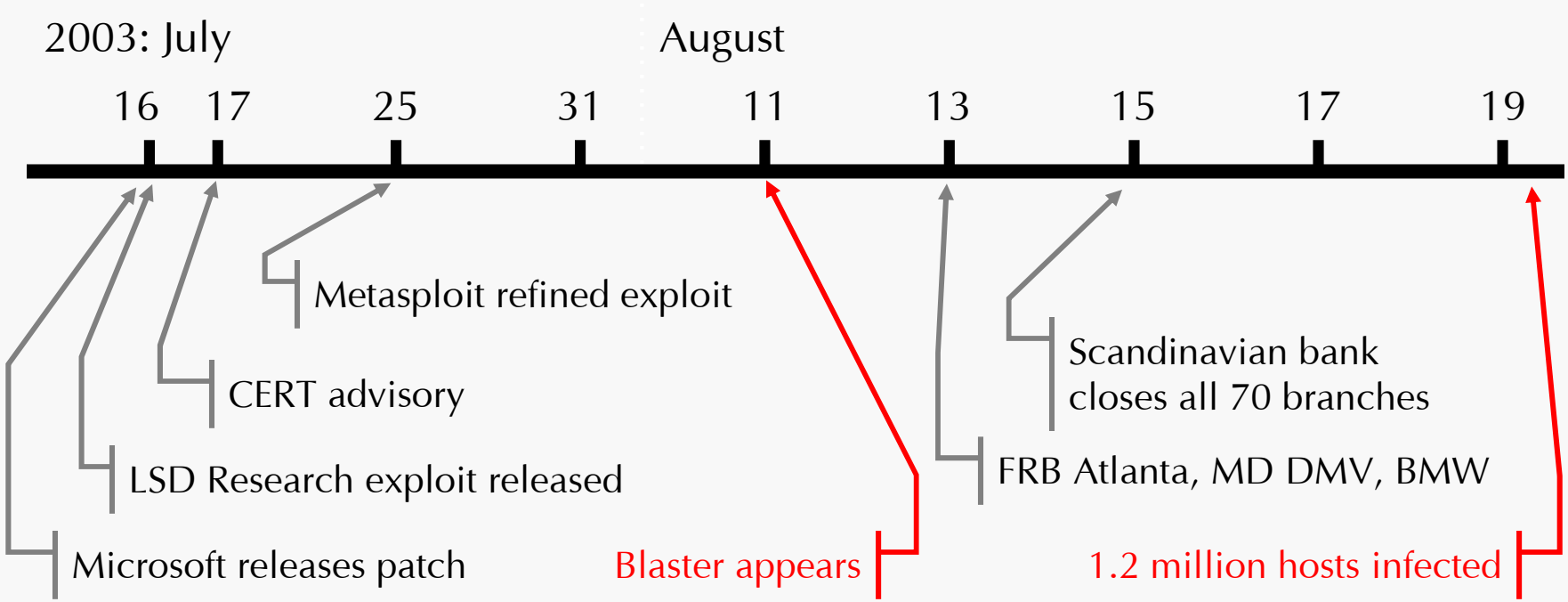
The Blaster Worm

- Multi-stage worm exploiting Windows vulnerability



The Blaster Worm

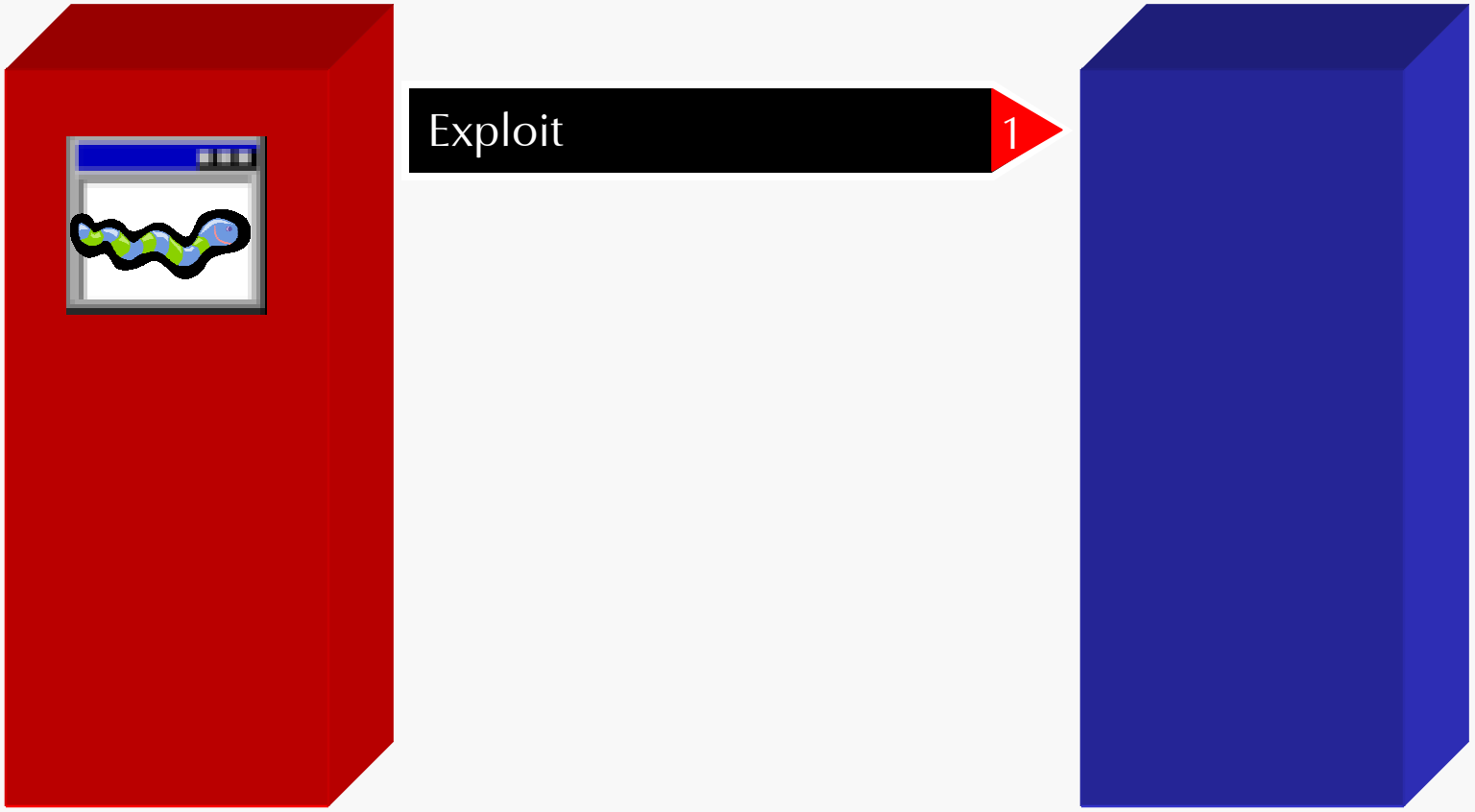
- Multi-stage worm exploiting Windows vulnerability



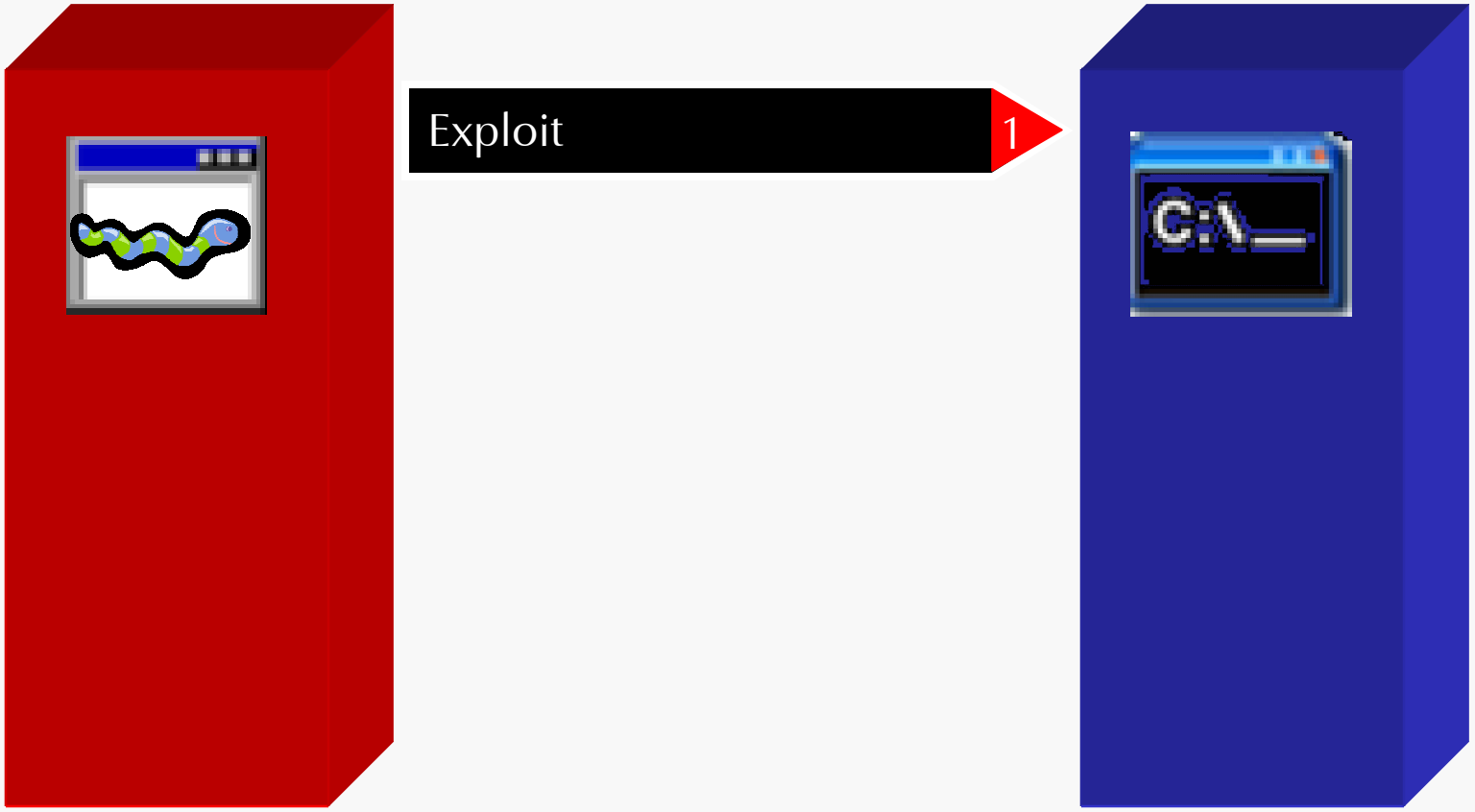
Blaster: Attack Vector

- Uses a Microsoft Windows RPC DCOM vulnerability.
- Coding flaw:
  1. The RPC service passes part of the request to function `GetMachineName()`.
  2. `GetMachineName()` copies machine name to a fixed 32-byte buffer.

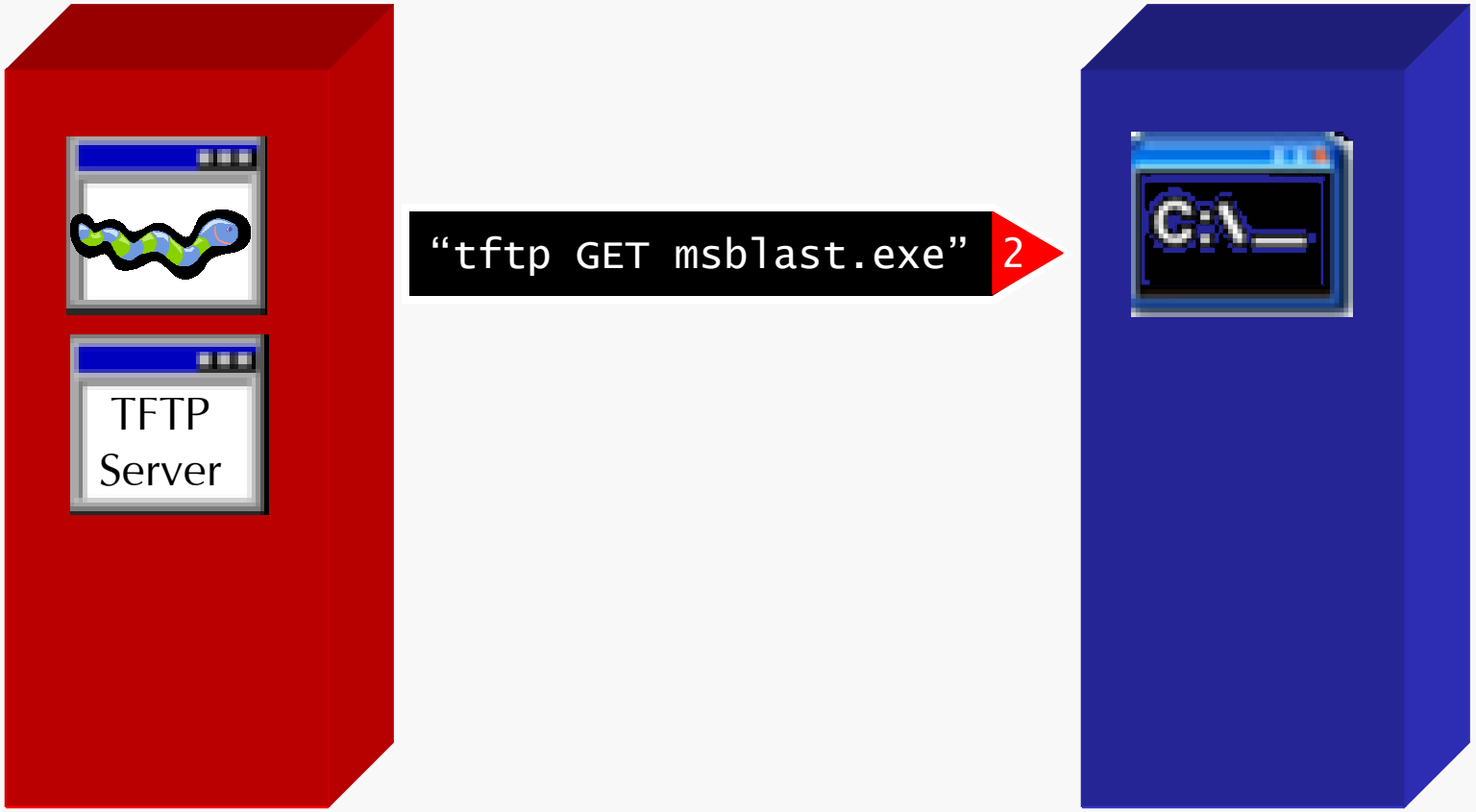
Blaster: Attack Vector



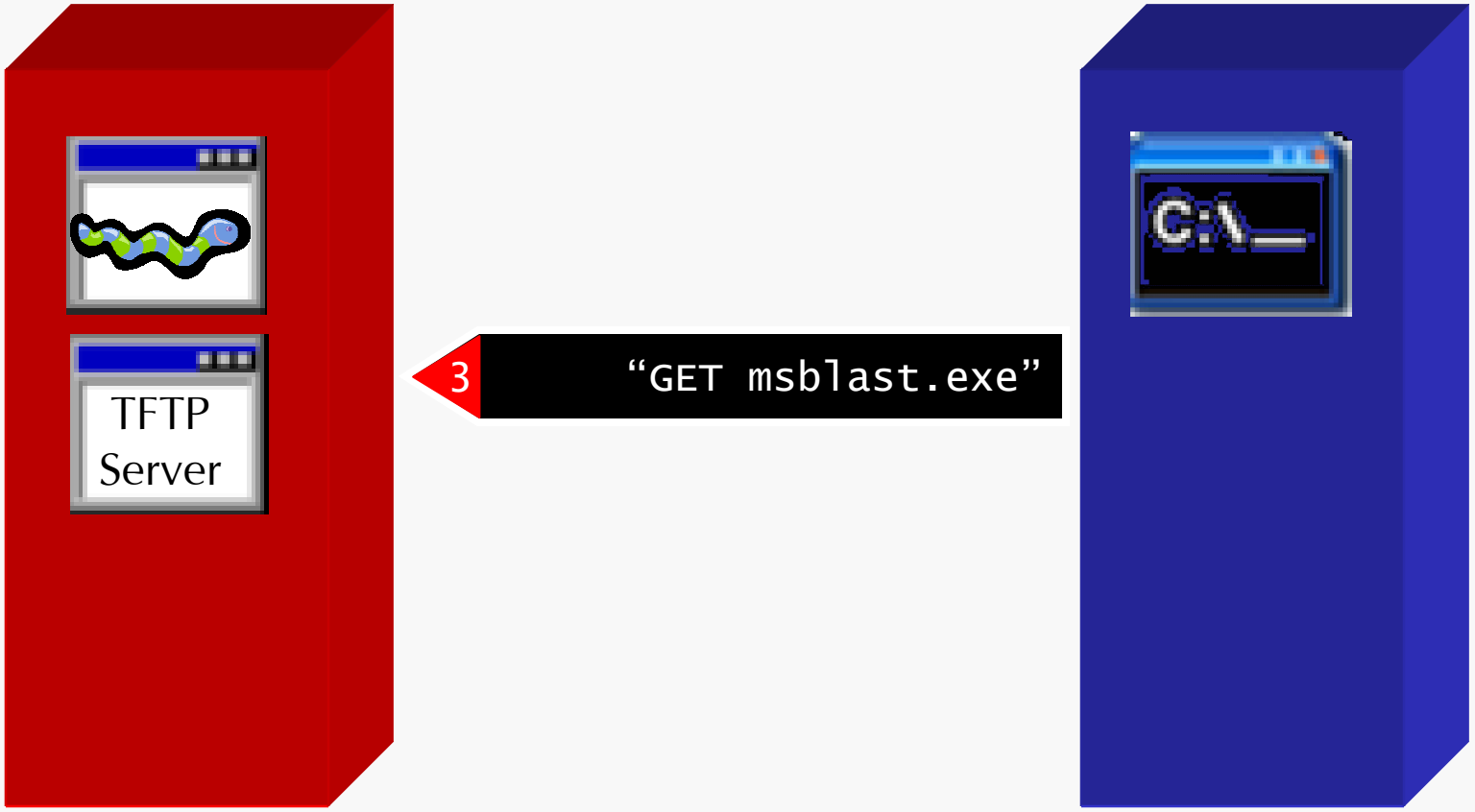
Blaster: Attack Vector



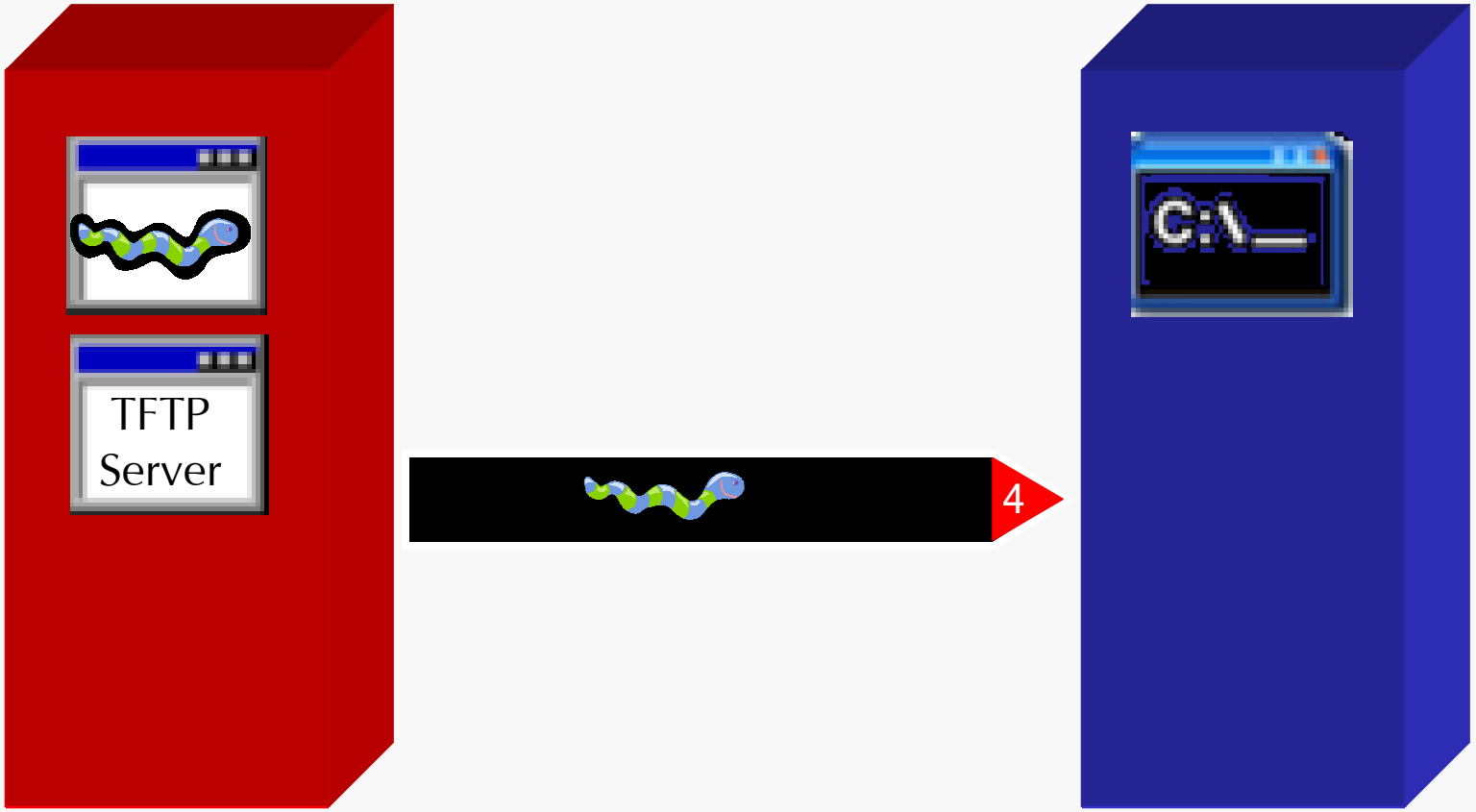
Blaster: Attack Vector



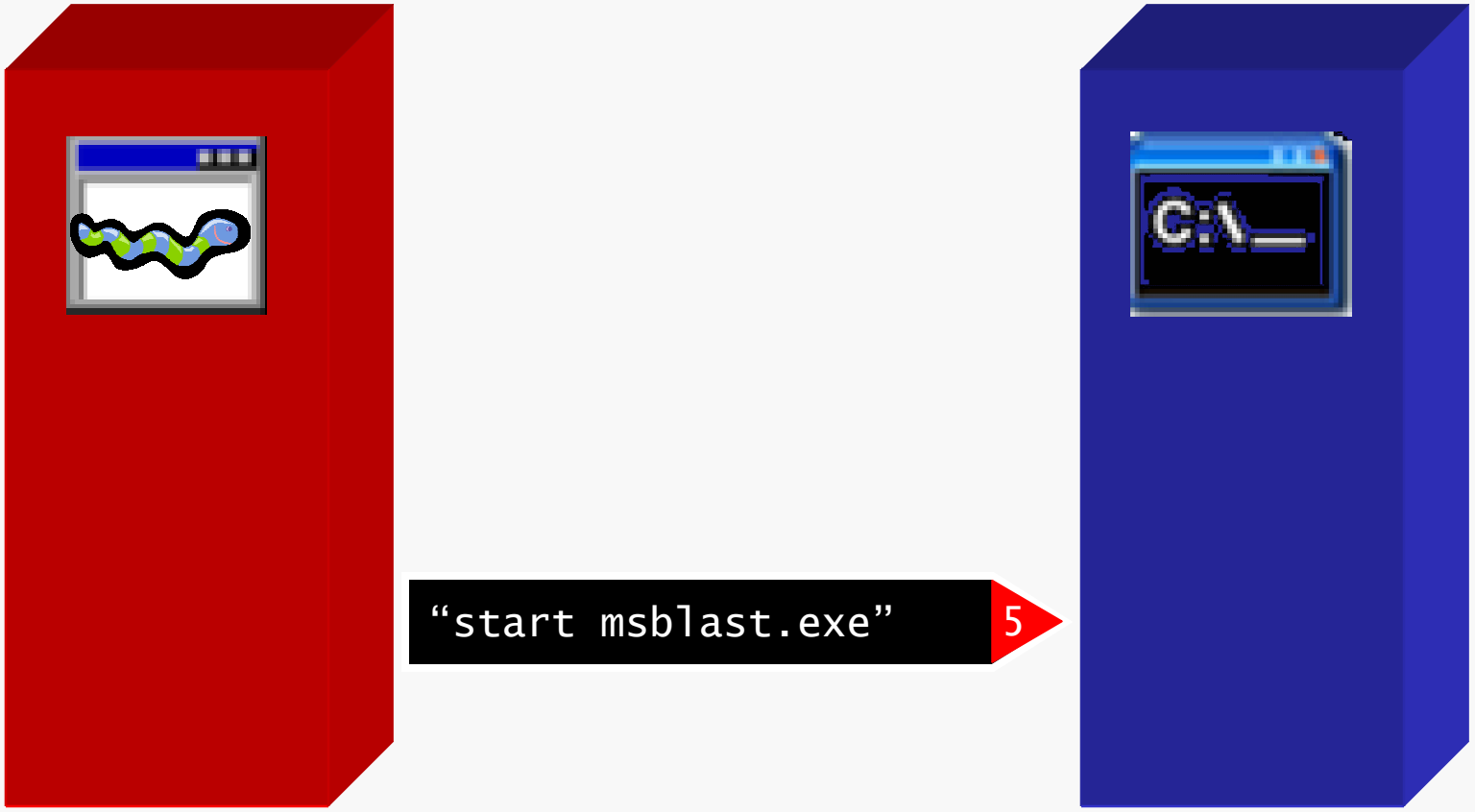
Blaster: Attack Vector



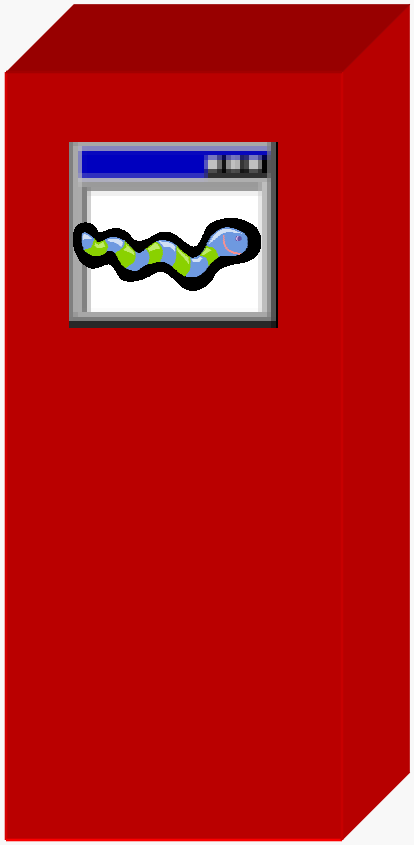
Blaster: Attack Vector

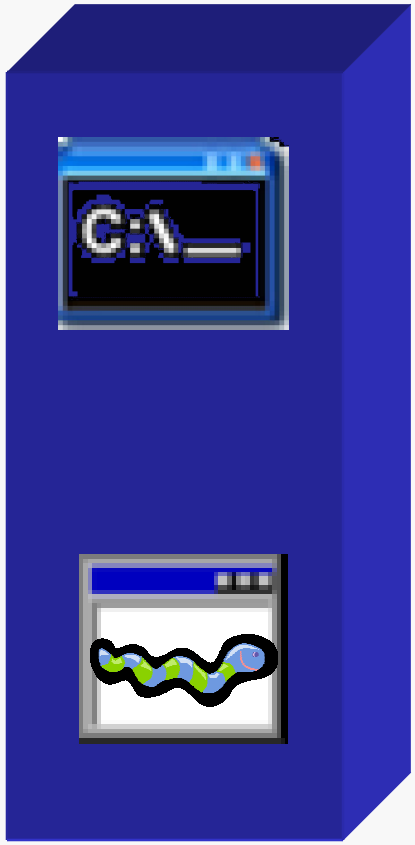


Blaster: Attack Vector



Blaster: Attack Vector





Blaster: Payload

- Worm installs itself to start automatically.
- All infected hosts perform DDoS against windowsupdate.com .
  - SYN flood attack with spoofed source IP,
    Aug 15 → Dec 31 and
    after the 15th of all other months.

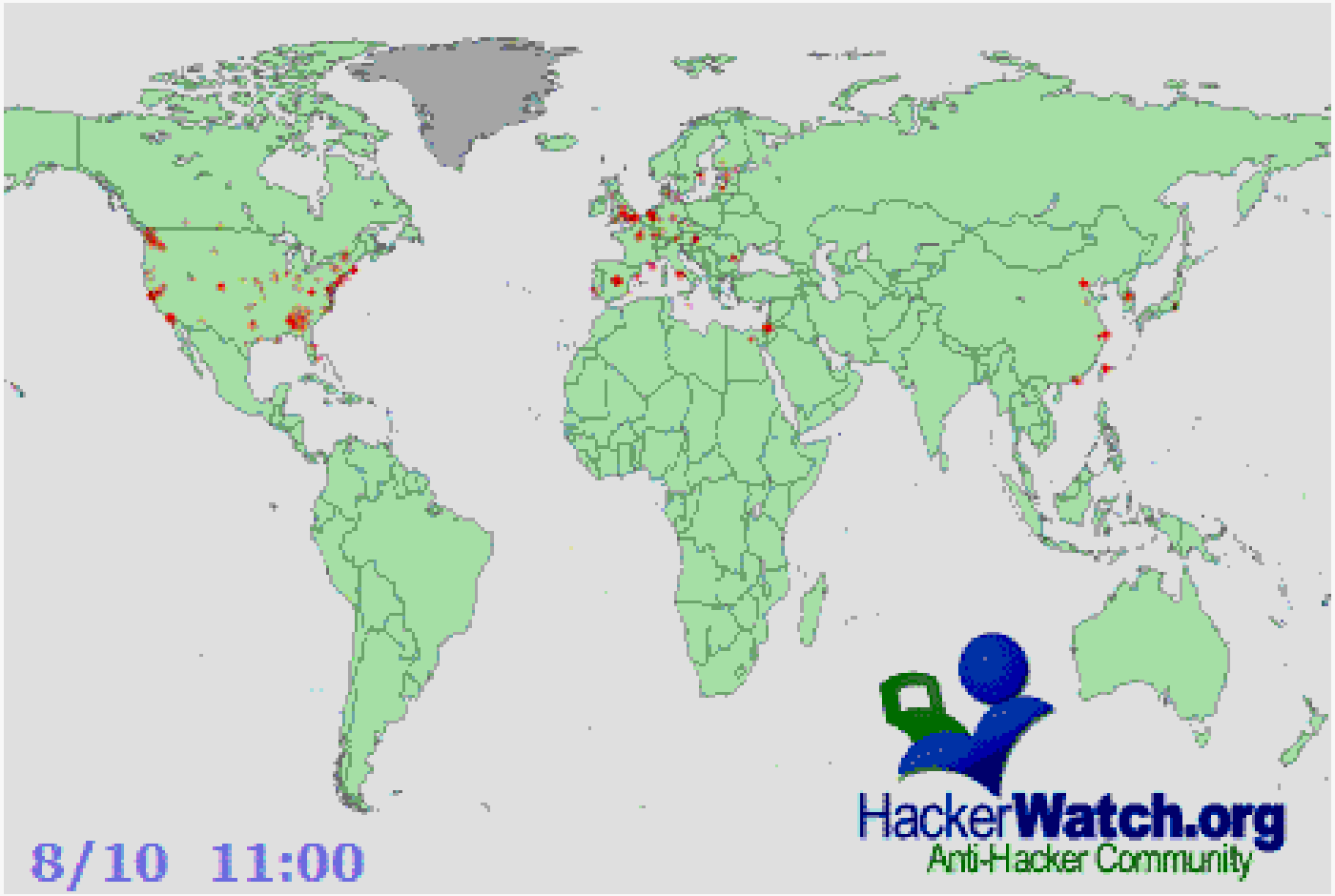
Blaster: Effect on Local Host

- RPC/DCOM disabled:
  - Inability to cut/paste.
  - Inability to move icons.
  - Add/Remove Programs list empty.
  - DLL errors in most Microsoft Office programs.
  - Generally slow, or unresponsive system performance.

Blaster: Spreading Algorithm

- Build IP address list:
  - 40% chance to start with local IP address.
  - 60% chance to generate random IP address.
- Probe 20 IPs at a time.
- Exploit type:
  - 80% Windows XP.
  - 20% Windows 2000.

Blaster: Infection Rate



Future Threat: Superworm

"Curious Yellow: the First Coordinated Worm Design" – Brandon Wiley

- Fast replication & adaptability:
  - Pre-scan the network for targets.
  - Worm instances communicate to coordinate infection process.
  - Attack vectors can be updated.
  - Worm code mutates.

Conclusions

- Vulnerabilities left unpatched can and will be used against you.
- Attackers are more sophisticated.
- Need to understand the attackers' perspective.

Malicious Code for Fun and Profit

Mihai Christodorescu

mihai@cs.wisc.edu

10 March 2005