



UC DAVIS



UnderStand

Unknown-vulnerability Collaborative Defense

# On Deriving Unknown Vulnerabilities from Zero-Day Exploits

August 29, 2005

*S. Felix Wu*

Computer Science Department  
University of California, Davis

<http://minos.cs.ucdavis.edu/>  
[wu@cs.ucdavis.edu](mailto:wu@cs.ucdavis.edu)

08/29/2005

LANOMS'05, Porto Alegre, Brazil

1

# WORM



UnderStand

- Since November 2<sup>nd</sup> of 1988...
  - Robert T. Morris, Code Red, Nimda, Slammer, Blaster, and many others...
- inject → infect → spread



# WORM

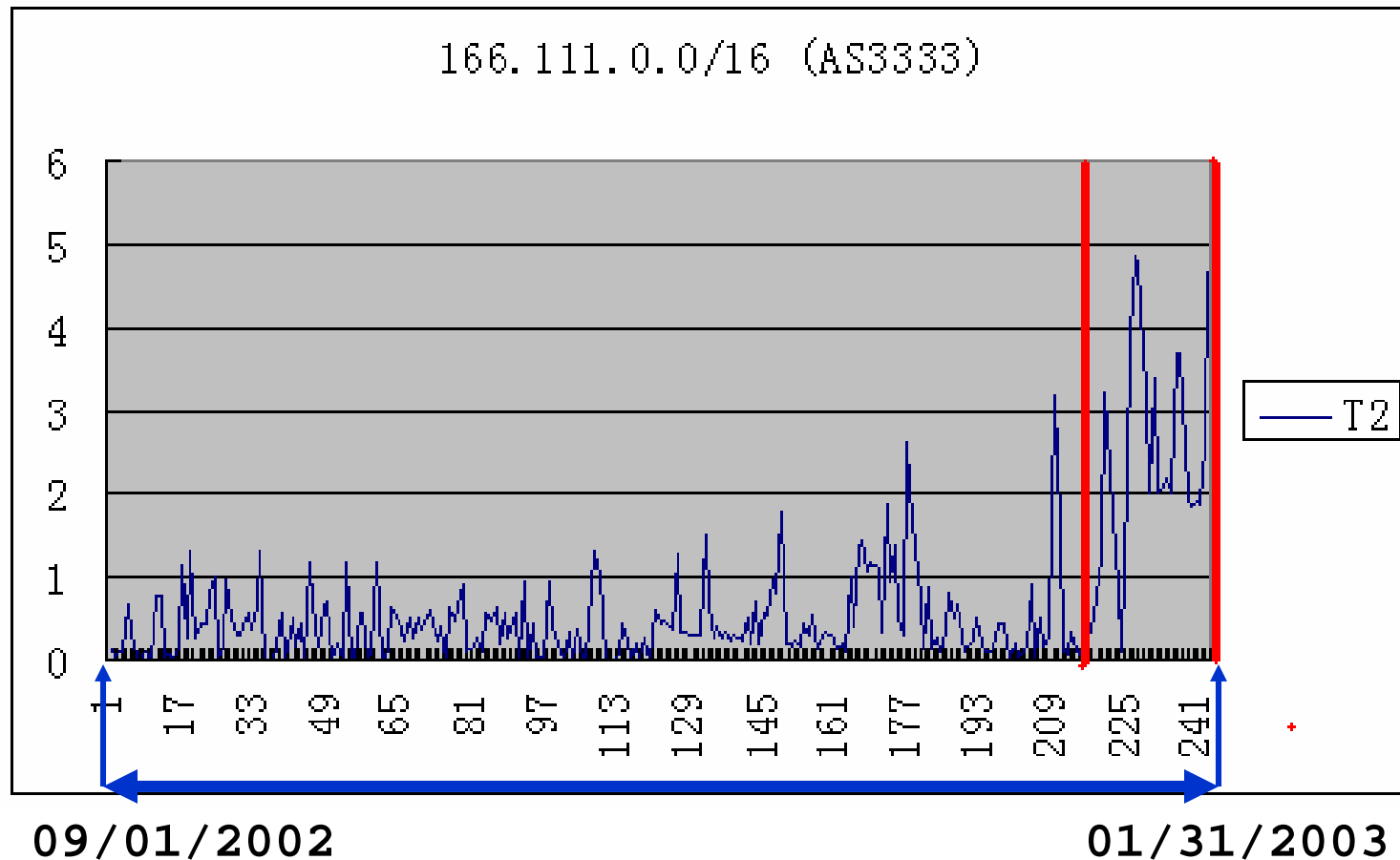
- Since November 2<sup>nd</sup> of 1988...
  - Robert T. Morris, Code Red, Nimda, Slammer, Blaster, and many others...
- inject → infect → spread
- WORM is causing Internet-wide instability.

# Slammer → BGP

*Internet routing stability analysis on a Beijing prefix*



UnderStand



# Network meets Software



UnderStand

- An interesting interaction among the Internet, the software on the hosts, and the worms themselves.
- The “short-term” Reality:
  - Estimated 40~50% of Internet hosts are still vulnerable to CodeRed.

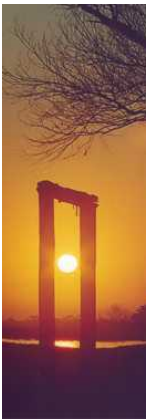


# WORM

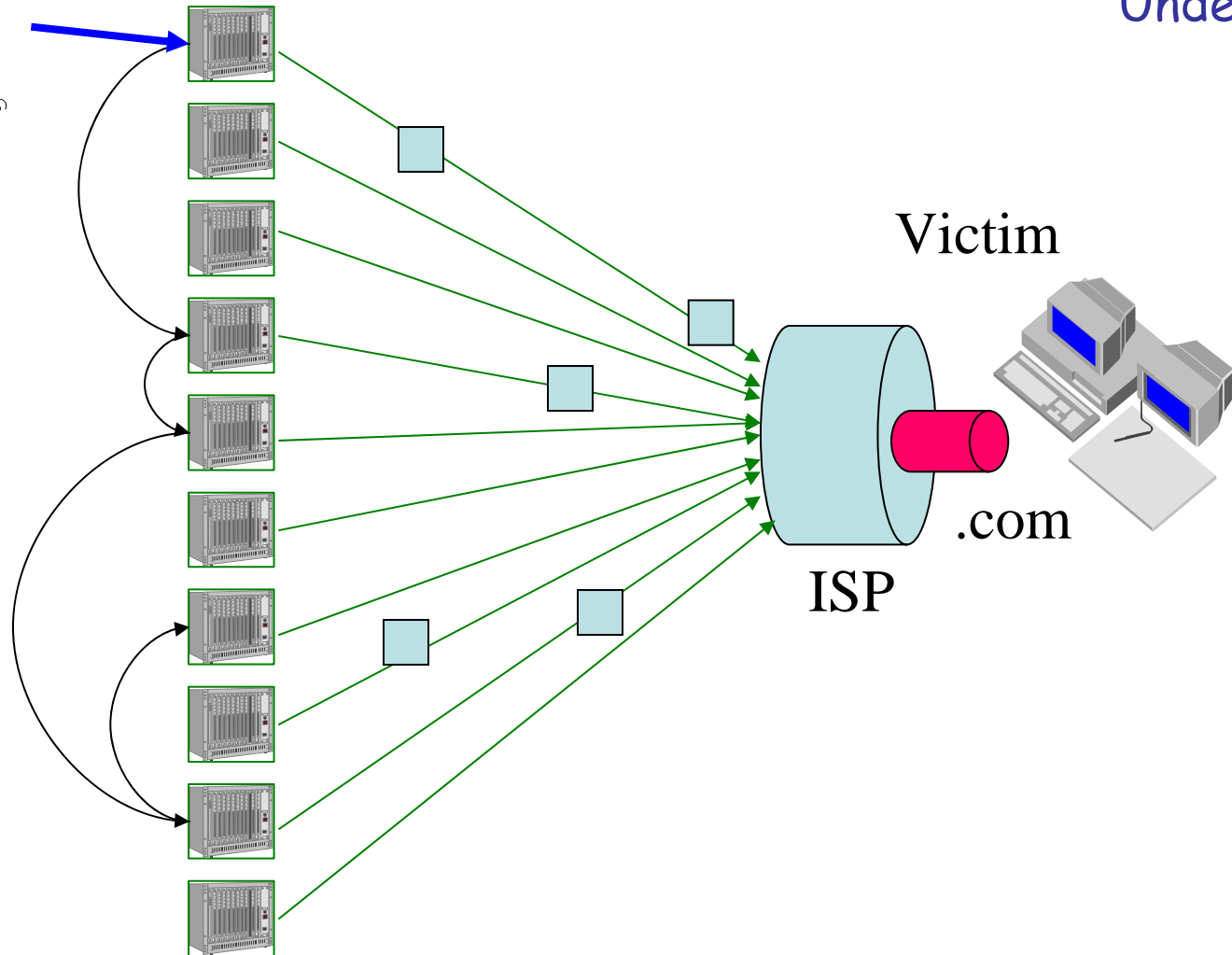
- Since November 2<sup>nd</sup> of 1988...
  - Robert T. Morris, Code Red, Nimda, Slammer, Blaster, and many others...
- inject → infect → spread
- WORM is causing Internet-wide instability.
- WORM is a critical first step for the attacker to quickly build the large-scale attacking infrastructure.



# WORM + DDoS



08/29/2005



LANOMS'05, Porto Alegre, Brazil



UnderStand

# They are getting better...

- The rapid evolution of the “attacker’s community”



08/29/2005

LANOMS'05, Porto Alegre, Brazil

8



## They are getting better...

- The rapid evolution of the “attacker’s community”
- And, many thanks to our rapid growing software industry in the past “N” years as well...



# Software Vulnerability

- Software vulnerabilities are weaknesses, being introduced during the “software engineering” process, that can potentially be exploited by attackers.
  - OS kernels, device drivers, applications...
- There are other types of vulnerabilities in our software systems that can be exploited.

# Software Vulnerability



- Difficulties in security management
  - we don't know how attackers are going to attack us,
  - And, we don't know which vulnerabilities can/will be exploited, either.

# Software Vulnerability



- Focus on Software Vulnerabilities
- Two approaches
  - better software engineering
  - better vulnerabilities understanding





# Software Vulnerability

- Focus on Software Vulnerabilities
- Two approaches
  - better software engineering

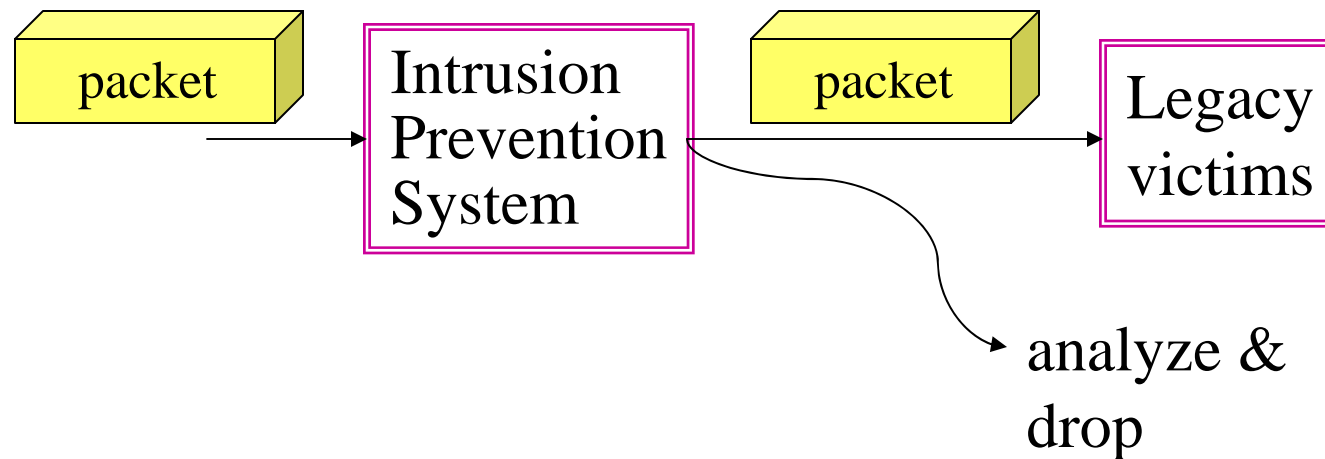
➡ **better vulnerabilities understanding**

Practically, around the Internet, we currently have and will still have a large number of legacy software systems around for “quite a while.”



# Network-based Solutions

- “Intrusion Prevention Systems” or “Advanced Firewalls”



# Vulnerability versus Exploit



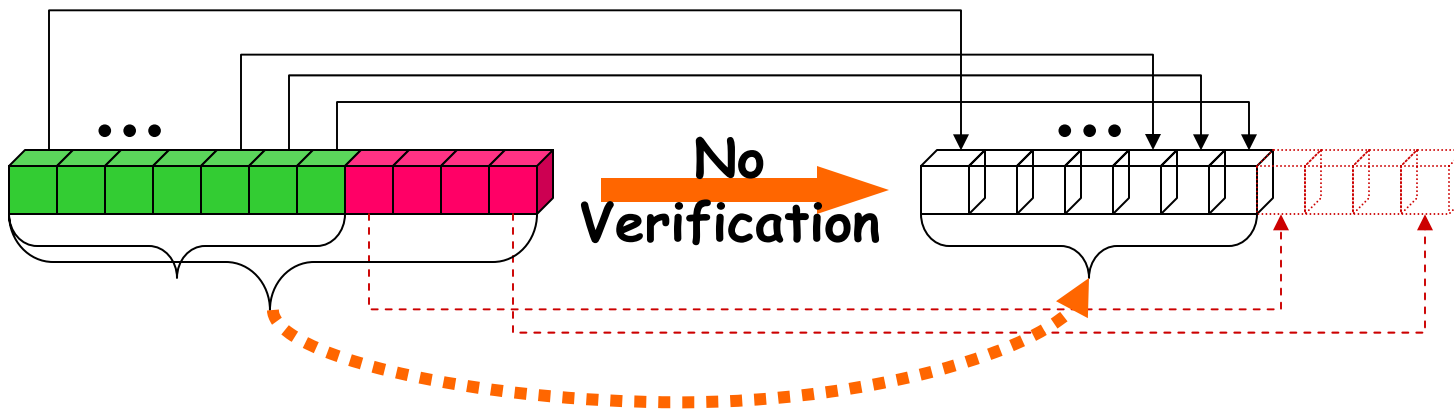
- Vulnerability
  - the “weak” points in the software
  - applications or even the kernel itself
  - **This talk → “control flow hijack” based on buffer overflow.**
- Exploit
  - the attack code utilizing one or more vulnerabilities



# Buffer Overflow

## Some unsafe functions in C library:

```
strcpy(char *dest, const char *src);
strcat(char *dest, const char *src);
getwd(char *buf);
gets(char *s);
fscanf(FILE *stream, const char *format, ...);
scanf(const char *format, ...);
realpath(char *path, char resolved_path[]);
sprintf(char *str, const char *format);
```

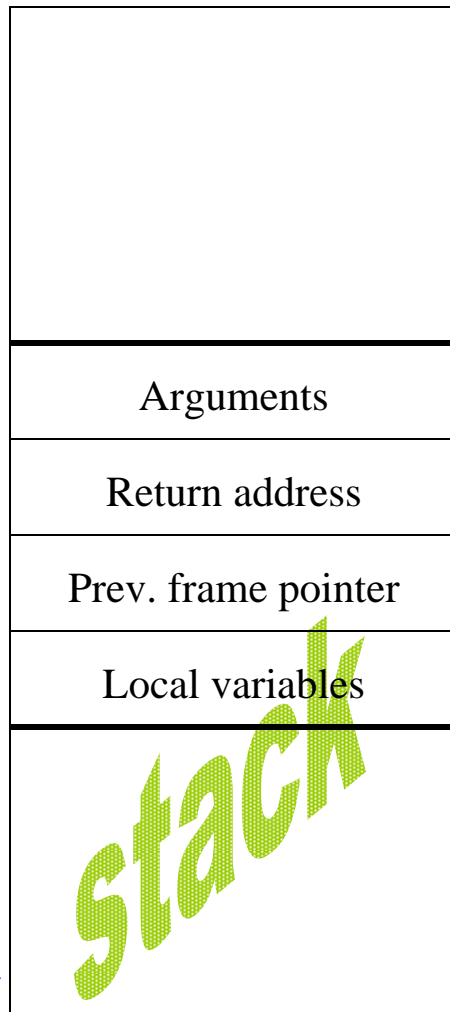




# Memory Structure



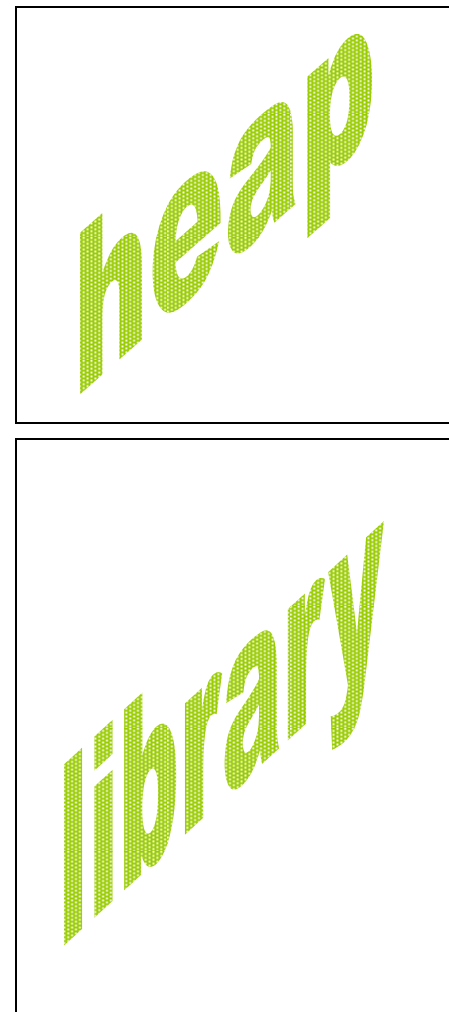
High



String  
Growth



Stack  
Growth



Stack  
Pointer

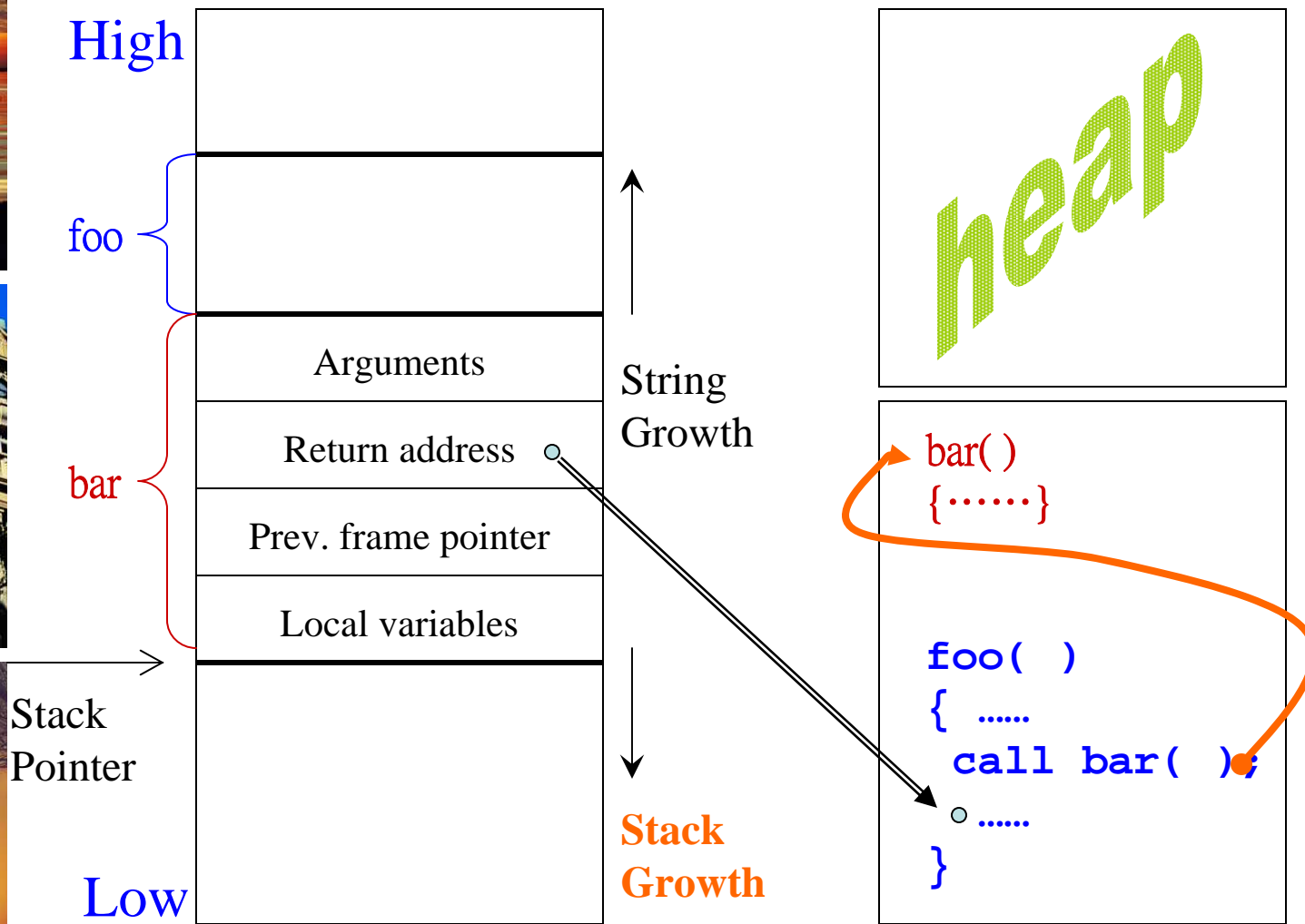
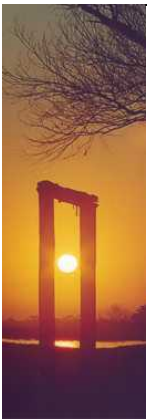
Low

08/29/2005

LANOMS'05, Porto Alegre, Brazil



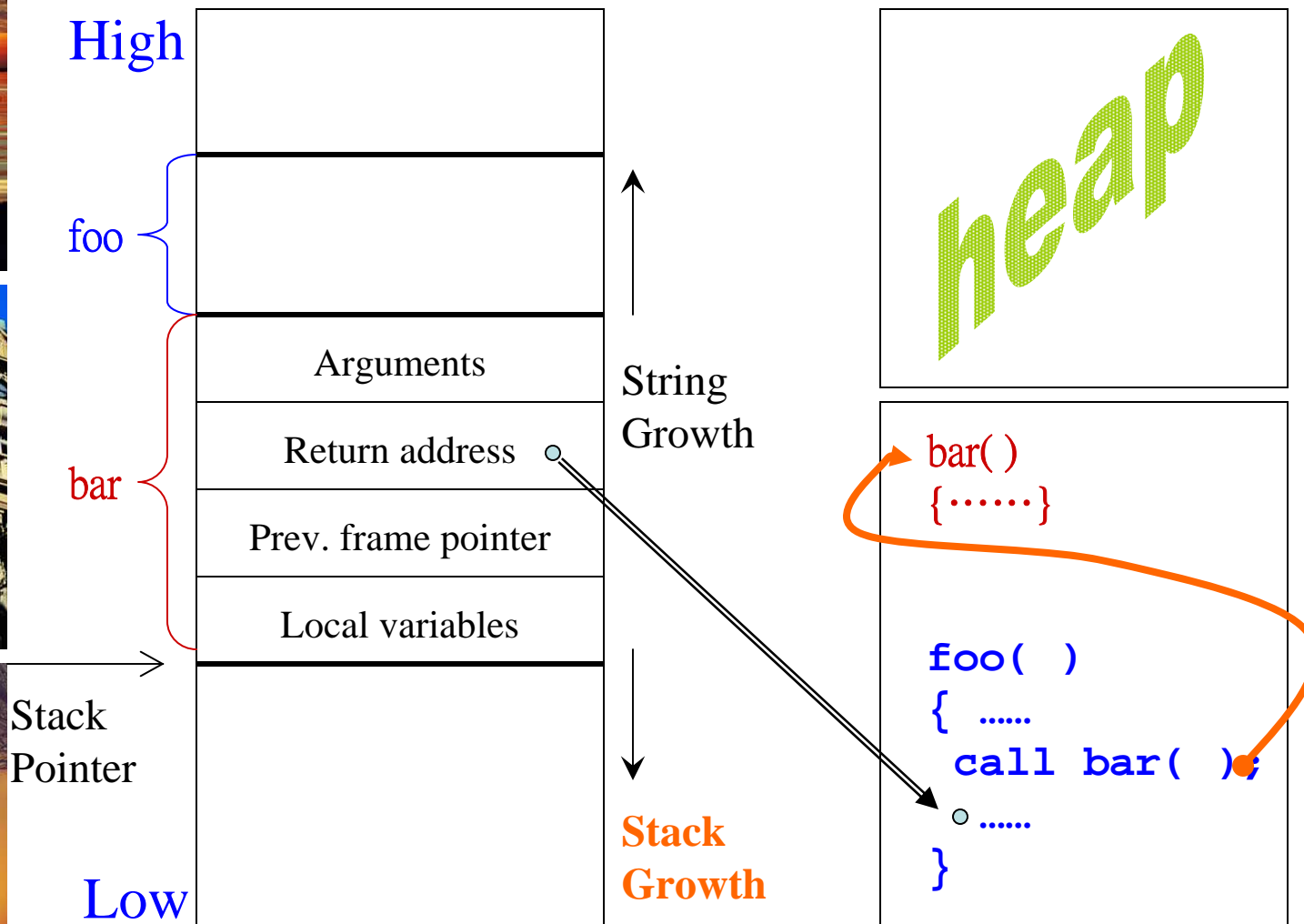
# Memory Structure





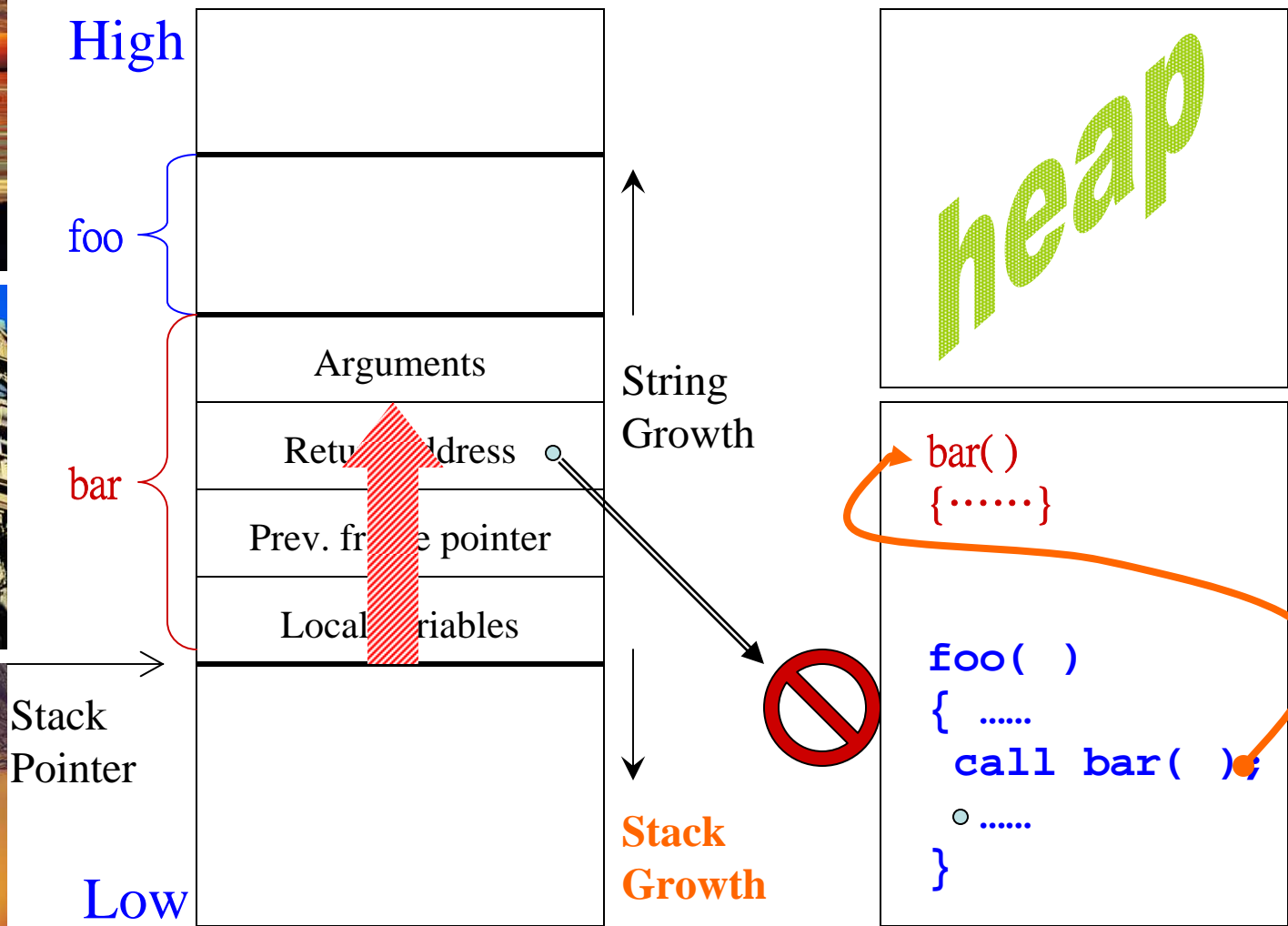
UnderStand

# Control Flow Hijack



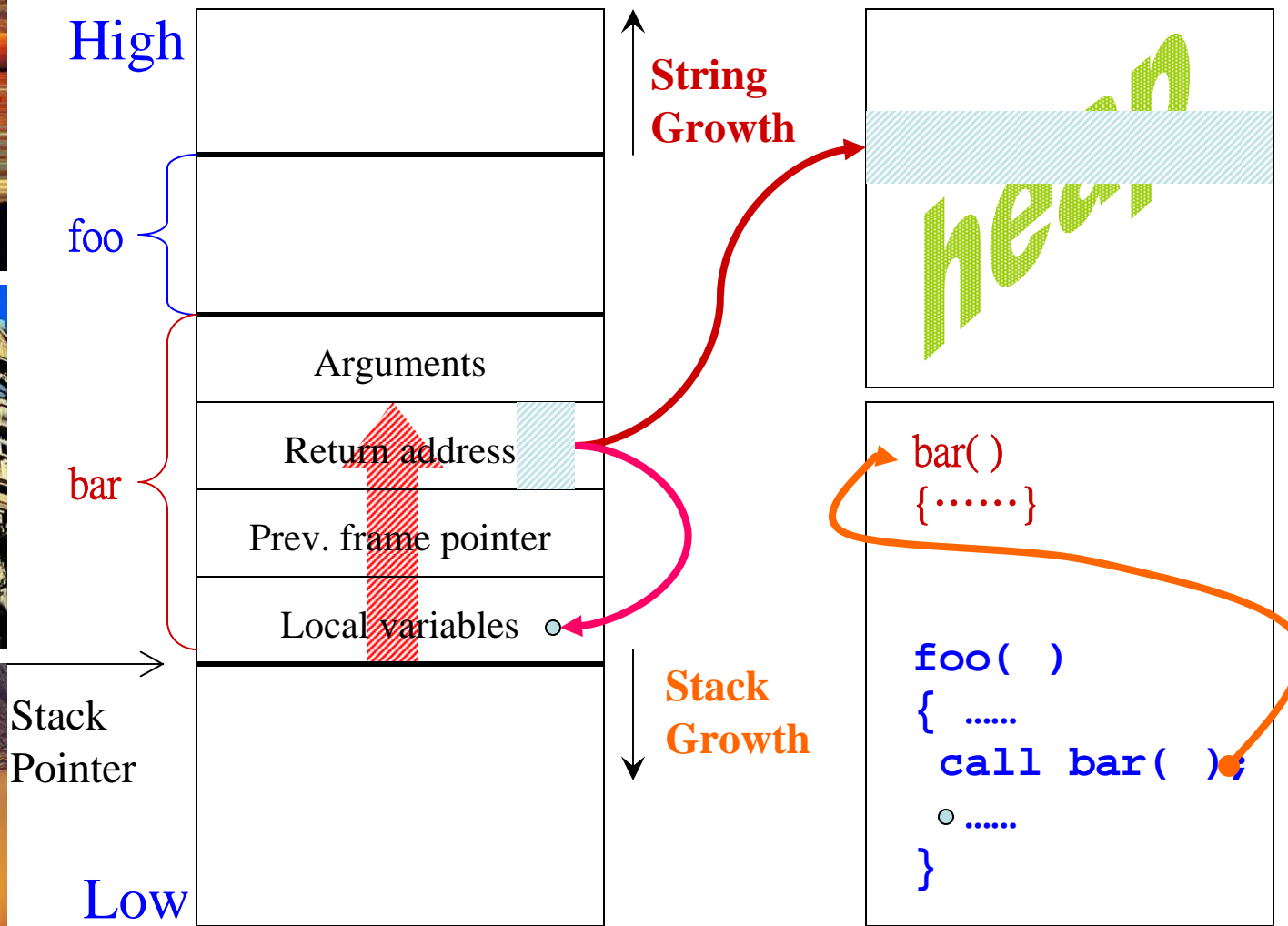


# Primitive → Hijack





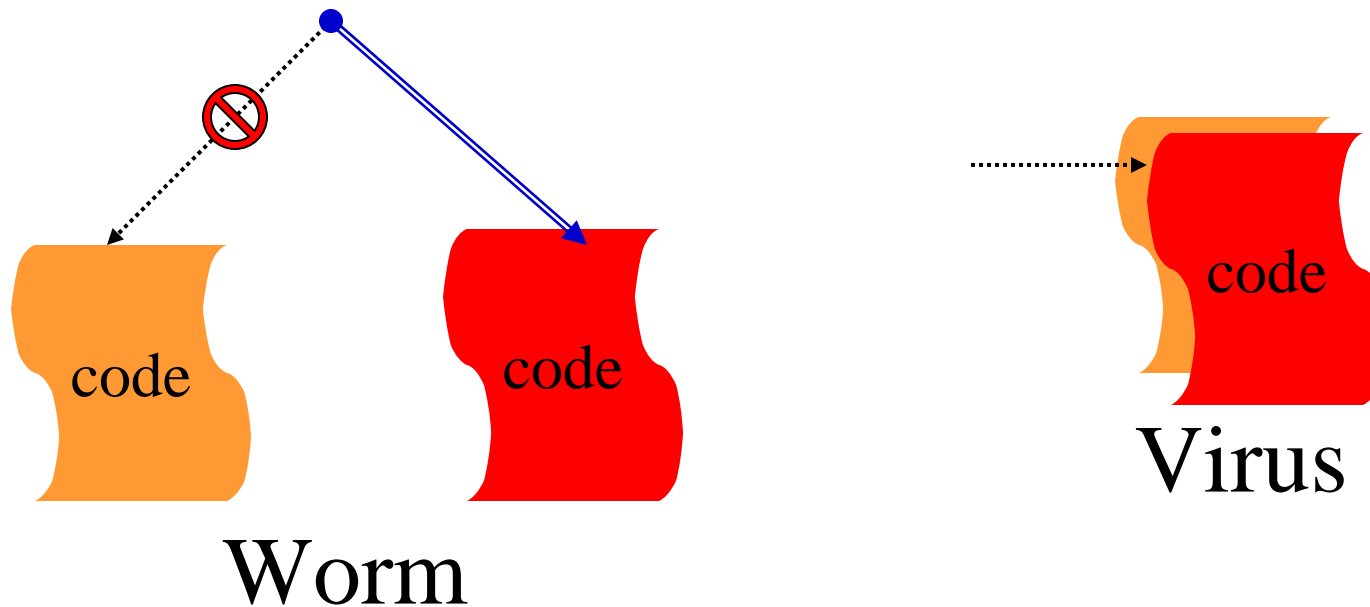
# Control Flow Hijack





# Control Flow Hijack

- I want “my code” executed!
  - Malicious code injection
  - Control flow redirection/hijacking





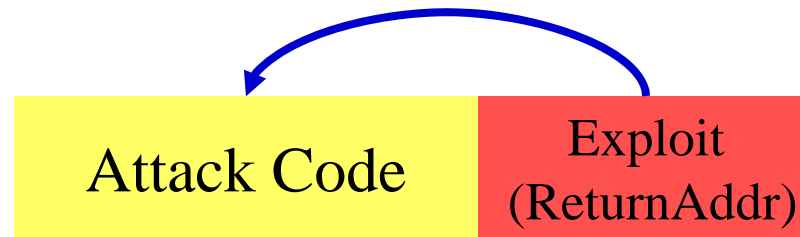
08/29/2005

# A Single Packet Exploit



UnderStand

Return Address == 0x4739a304





UnderStand

# Example



```

0000000 9090 9090 9090 9090 9090 9090 9090 9090
*
00001f0 9090 9090 22eb 895e 89f3 83f7 07c7 c031
0000200 89aa 89f9 abf0 fa89 c031 b0ab 0408 cd03
0000210 3180 89db 40d8 80cd d9e8 ffff 2fff 6962
0000220 2f6e 6873 f822 bfff f822 bfff f822 bfff
0000230 f822 bfff f822 bfff f822 bfff f822 bfff
*
00004a0 f822 bfff f822 bfff f822 bfff 9090 9090
00004b0 fa48 bfff
  
```



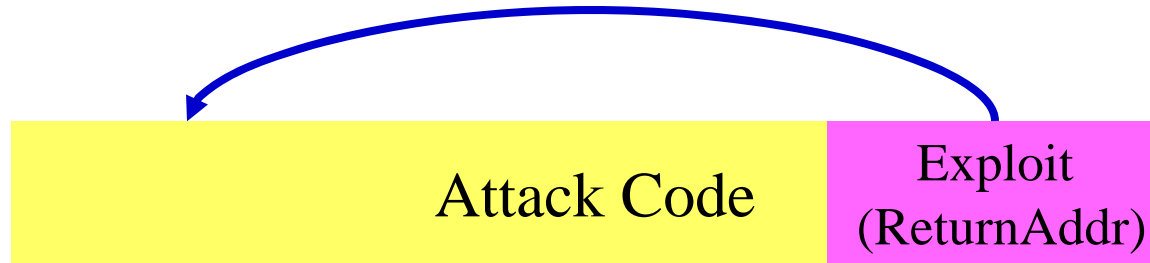
UnderStand

# Example: NOP-sled

0000000	9090	9090	9090	9090	9090	9090	9090	9090	9090
*									
00001f0	9090	9090	22eb	895e	89f3	83f7	07c7	c031	
0000200	89aa	89f9	abf0	fa89	c031	b0ab	0408	cd03	
0000210	3180	89db	40d8	80cd	d9e8	ffff	2fff	6962	
0000220	2f6e	6873	f822	bfff	f822	bfff	f822	bfff	
0000230	f822	bfff	f822	bfff	f822	bfff	f822	bfff	
*									
00004a0	f822	bfff	f822	bfff	f822	bfff	9090	9090	
00004b0	fa48	bfff							

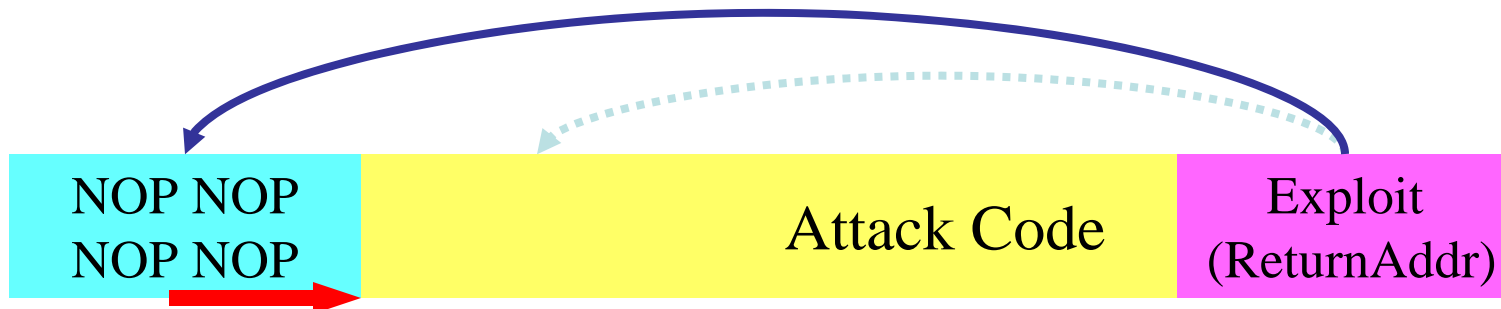
Sometime we can not easily determine the “exact” memory address to jump into...

# “NOP Sled” Engineering



```
code[] = "\xeb\x2a\x5f\xc6\x47\x07\x00\x89\x7f\x08\xc7\x47";
strcpy(buf, code);
```

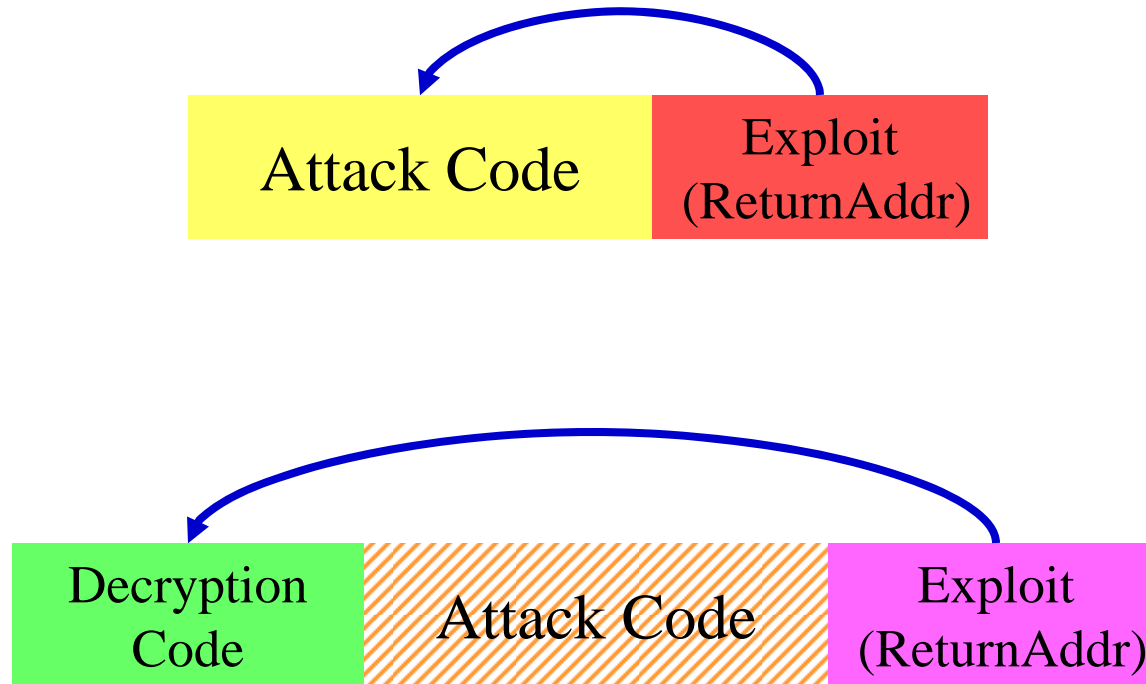
```
buf = "\xeb\x2a\x5f\xc6\x47\x07"
```



And, sometimes, we simply want to find a way to avoid “\x00”.



attack polymorphism  
(many different ways)



**The Signature Explosion Problem!!**

# Vulnerability versus Exploit



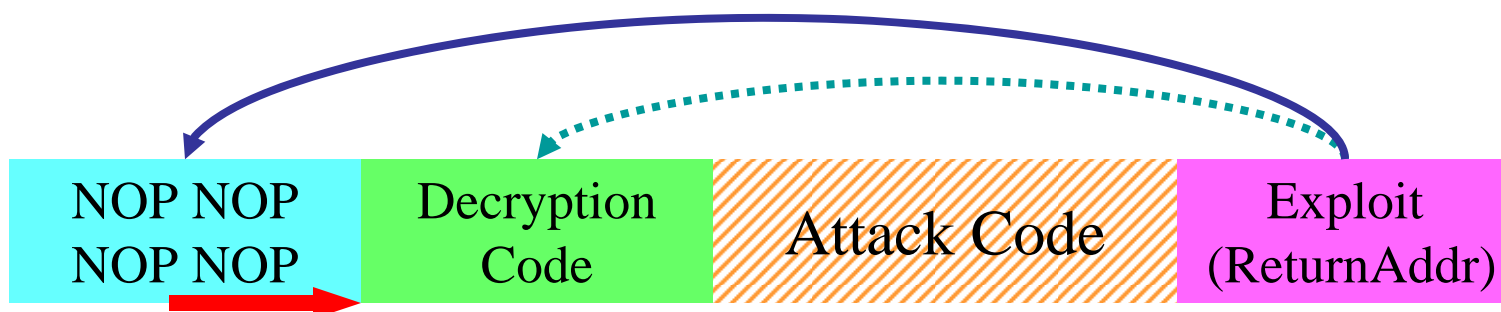
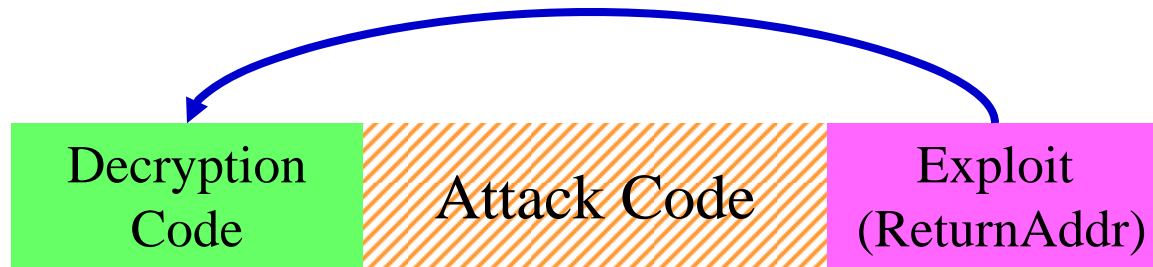
UnderStand

- $1 \leftrightarrow M$  or  $N \leftrightarrow M$
- Polymorphic tools available
  - A Naïve approach:  $M \rightarrow \infty$
- Can we find the “invariants”?
  - We need to avoid “signature explosion”...





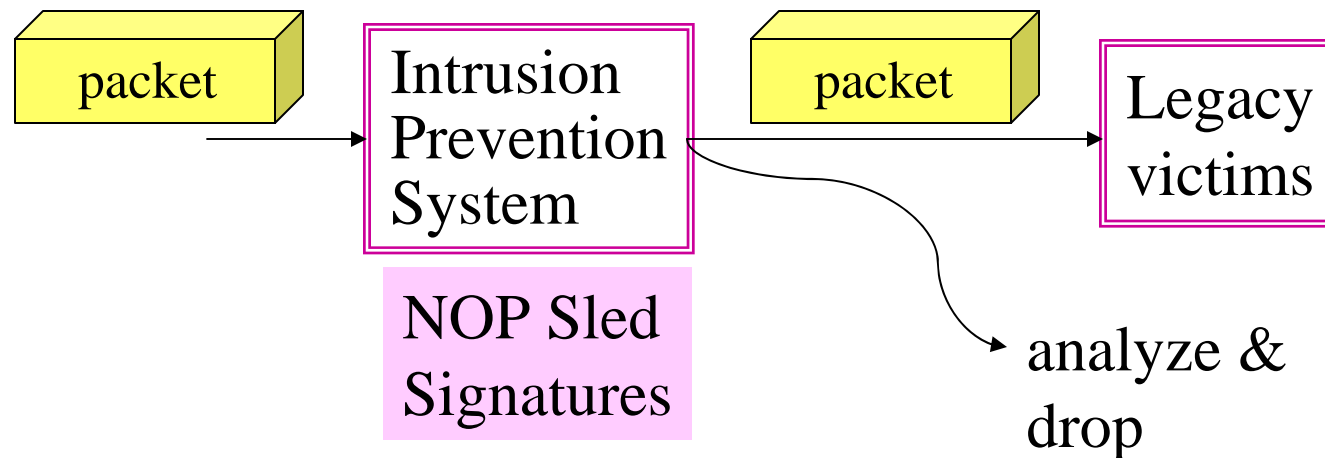
UnderStand





# Detecting “NOP Sleds”

- “Intrusion Prevention Systems” or “Advanced Firewalls”



# A WORM with a NOP-Sled



UnderStand

0000000	9090	9090	9090	9090	9090	9090	9090	9090	9090
*									
00001f0	9090	9090	22eb	895e	89f3	83f7	07c7	c031	
0000200	89aa	89f9	abf0	fa89	c031	b0ab	0408	cd03	
0000210	3180	89db	40d8	80cd	d9e8	ffff	2fff	6962	
0000220	2f6e	6873	f822	bfff	f822	bfff	f822	bfff	
0000230	f822	bfff	f822	bfff	f822	bfff	f822	bfff	
*									
00004a0	f822	bfff	f822	bfff	f822	bfff	9090	9090	
00004b0	fa48	bfff							



UnderStand

# A Polymorphic WORM



0000000	5247	5237	5759	9199	984e	602f	4b58	9555
0000010	3792	4997	6059	5a5d	979c	9199	9242	9349
0000020	495e	5b37	4740	5d4f	4f99	975f	4492	3797
0000030	4297	9e93	4598	404a	9696	4652	5150	5e4f
0000040	454d	99fc	5251	5042	9b37	4042	4a95	4459
0000050	4592	4998	935f	275f	985d	f84e	4991	fc96
0000060	9796	4637	5b3f	9751	9754	9f5a	9543	4c9e
0000070	4740	9c96	499f	5652	934e	5355	479b	91f8
0000080	48fc	5d60	4742	9755	4450	4441	4697	5697
0000090	5b52	494f	434d	5899	f827	9957	4346	9796
00000a0	404c	4a45	6040	404c	4957	5798	99f9	569b
00000b0	4145	96fc	5140	4c56	f946	9348	4f4d	f8f8
00000c0	2f59	4c46	9647	4747	9e48	5137	4142	5b4d
00000d0	545f	55f9	5e56	4191	9249	519e	559e	6099



UnderStand

## NOP sleds

- “NOP sled” can/will NOT be a useful signature in detecting future WORMs...
- 80~90% of the WORMs today don't really need “NOP sleds” but, historically, they are still “left” there.





UnderStand

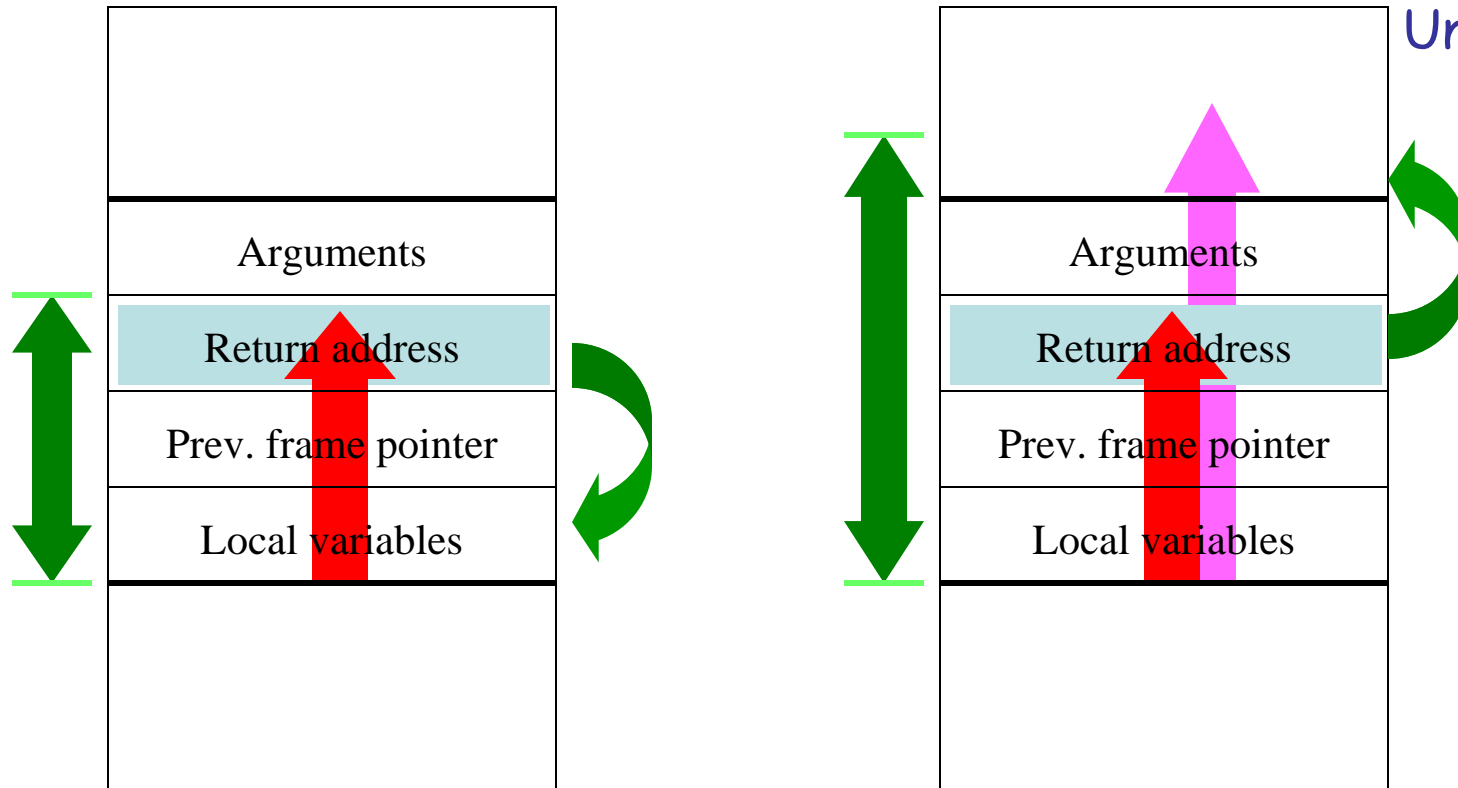
# BUTTERCUP

- Ideas:
  - Given a software exploit, the hacker can encrypt the malicious code but not the “hijacking” entry point (e.g., return address).
  - The hacker can twist the “return address” but practically not infinitely
    - **a range of memory addresses.**

# Memory Address Ranges



UnderStand



One “Exploit” has one “return address” value, but another exploit based on the same vulnerability might be using a different return address.

# size, offset and depth

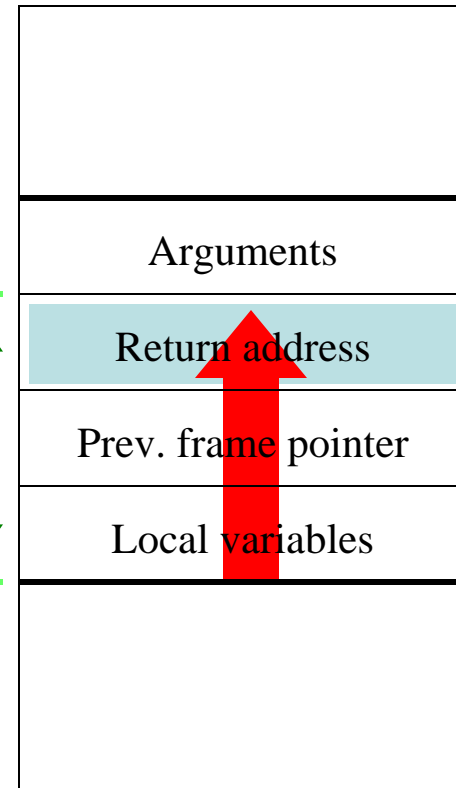


UnderStand

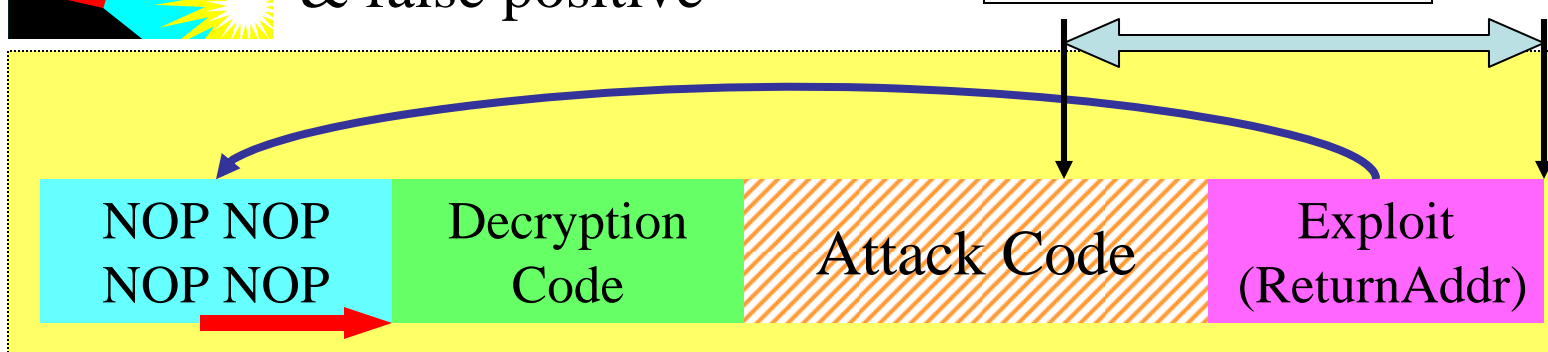
Is this packet a Slammer worm or a suspect "utilizing" the same vulnerability?

0x42b0caa4

0x42b0c914

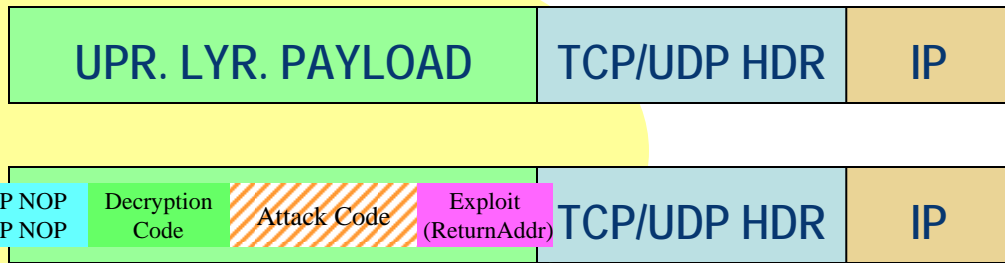


performance & false positive

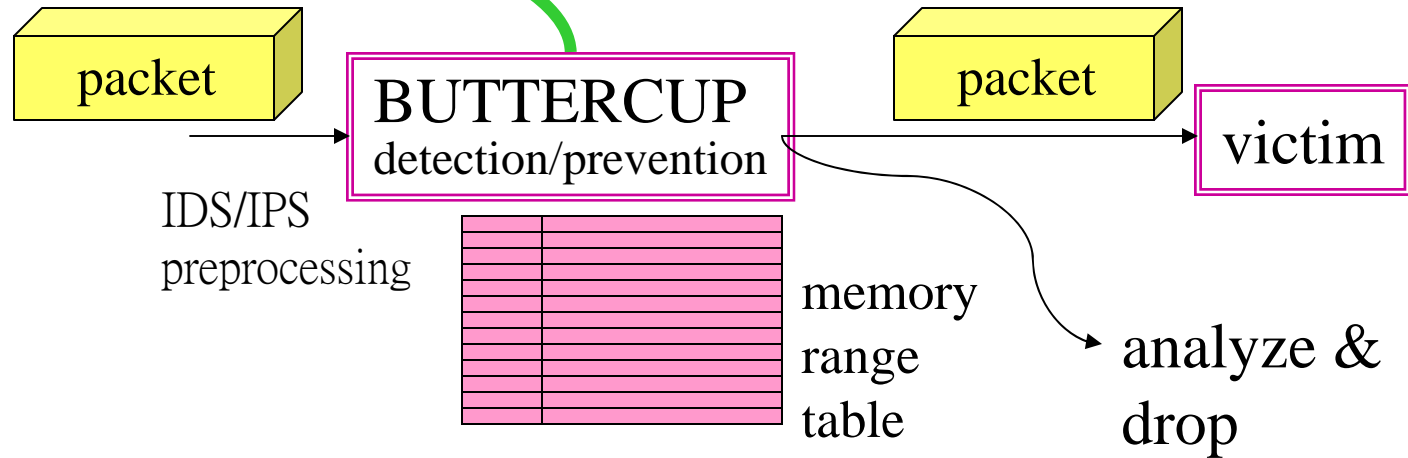




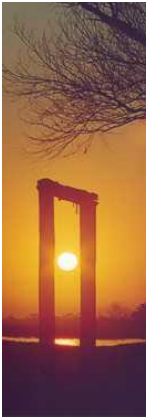
UnderStand



False  
Positive  
??



19 known exploits/vulnerabilities





TCPdump files	Snort versions				
	BC-range	BC-range- dsize<>	BC-range- dsize<>-RO-RD	BC-range- dsize>	BC-range- dsize>-RO-RD
Inside.tcpdump-00	0.6664	0.0144	0.0138	0.5293	0.2468
Outside.tcpdump-00	0.7276	0.0245	0.0249	0.5987	0.2833
sampledata01-dump	0.2203	0	0	0.2203	0.0275
tcpdins-00	0.3113	0.0057	0.0051	0.2408	0.1039
tcpdwk1mon-98	1.1237	0.0077	0.0093	0.9642	0.3240
tcpdwk1tue-98	1.0721	0.0075	0.0092	0.9275	0.3229
tcpdwk2wed-98	0.7887	0.0067	0.0059	0.6779	0.2592
tcpdwk2thu-98	0.9867	0.0153	0.0110	0.8788	0.3857
tcpdwk2fri-98	0.2995	0	0	0.2804	0.0539
tcpdinswk1mon-99	0.5048	0.0138	0.0132	0.4020	0.1916
tcpdinswk1tue-99	0.5788	0.0122	0.0117	0.4210	0.2202
tcpdinswk1wed-99	0.5210	0.0106	0.0099	0.4083	0.1902
tcpdinswk2mon-99	0.4927	0.0107	0.0098	0.3845	0.1710



UnderStand

about 30~180 days

In July, 2002 Microsoft announced the vulnerabilities!

On January 25, 2003 05:30 UTC, slammer was out!



We had about 6 months back then!!



BUTTERCUP, a network based approach, might have been more practical and scaleable than Windows Update!!

# Limitation

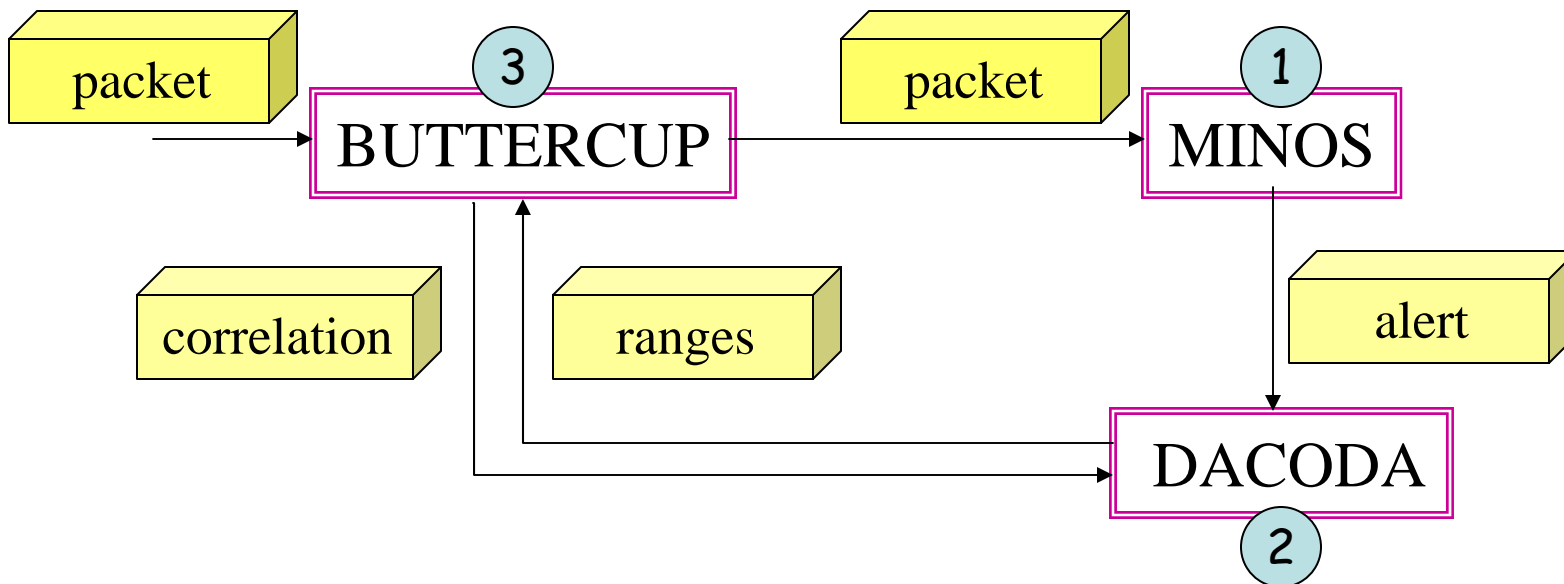


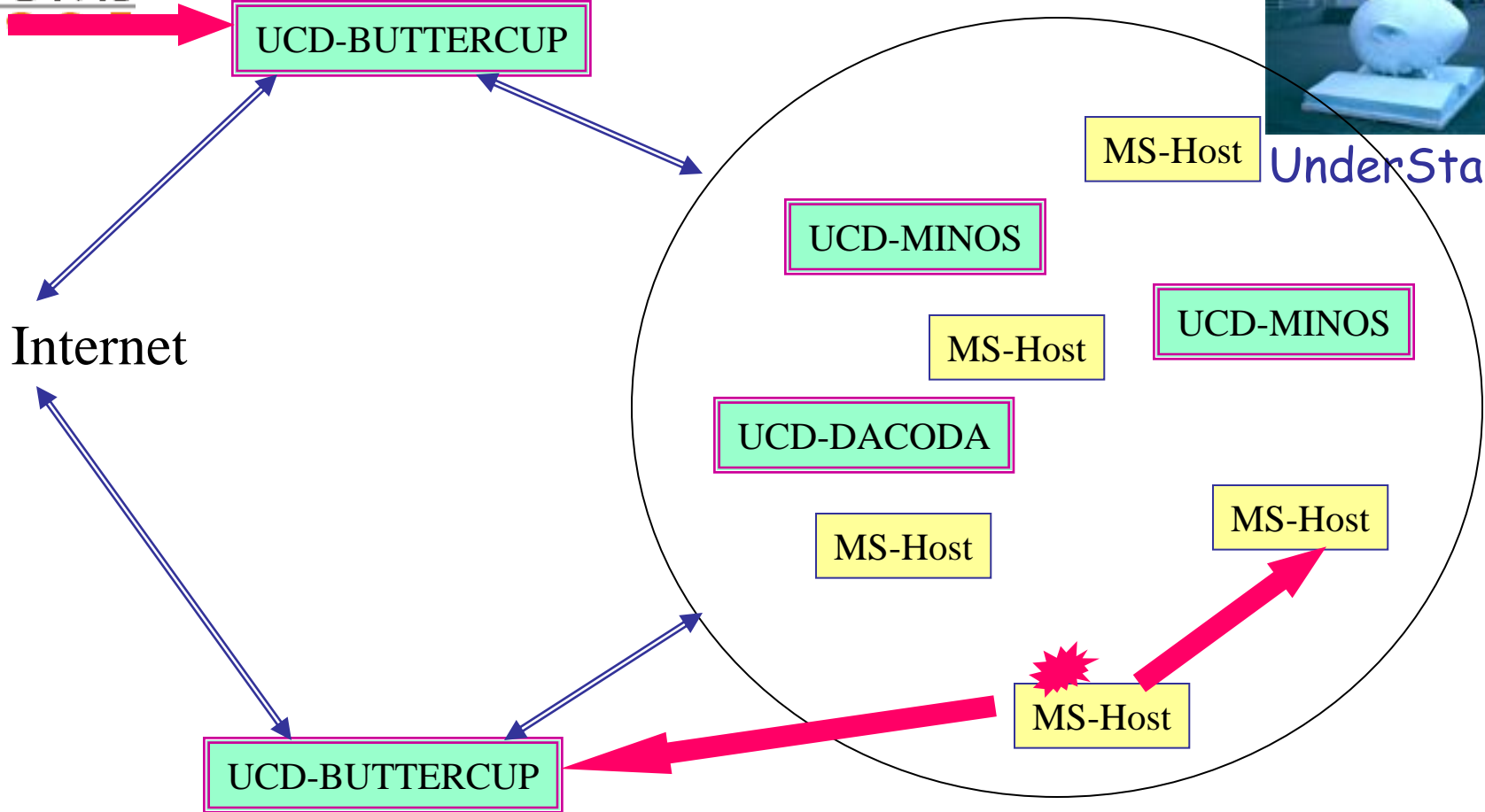
UnderStand

- BUTTERCUP will only work for “known vulnerabilities”!
  - But, it may work for Zero-day exploits based on known vulnerabilities.



# Unknown-vulnerability Collaborative Defense (UCD)





It is not perfect, but it is a reasonable & practical trade-off between large-scale system administration (manageability) and security!

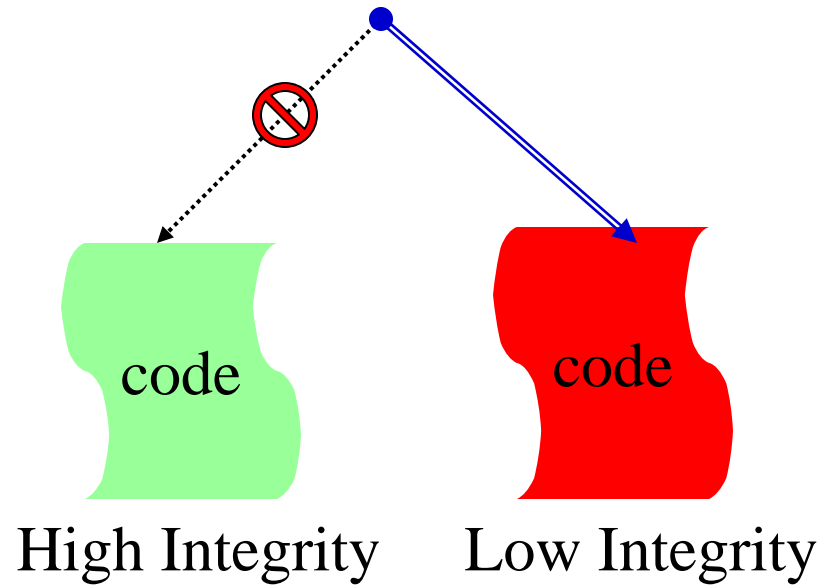
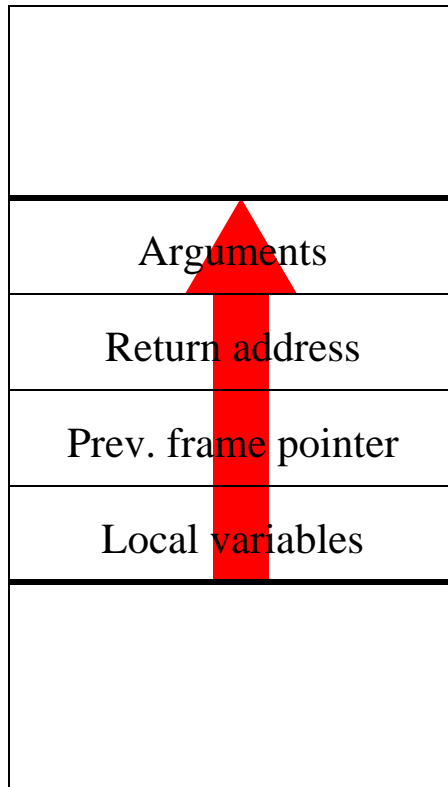
# Minos



UnderStand

- Can we detect “vulnerabilities/exploits” **with only one signature/invariance** at run-time?
  - We are not performing analysis on source code!
    - we might not have it, and, it might be too late.
  - “Run-Time” system monitoring
  - Anomaly detection → False Positive/Negative
- Detecting “*Control Flow Hijacks*”
  - from the CPU point of view
- A surprisingly simple practical solution with “**zero false positive**” ... at least empirically.

# Control Flow Hijack

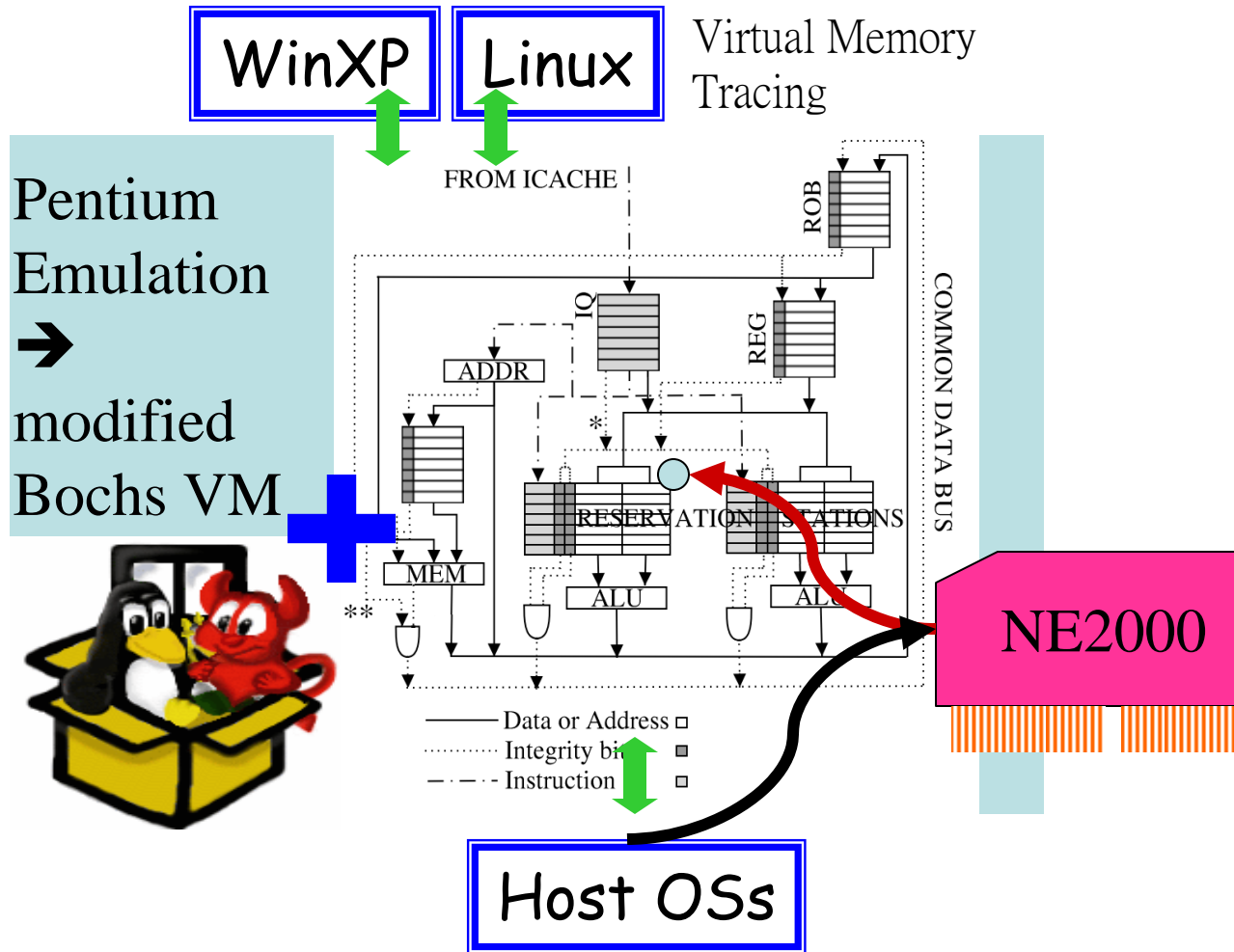




# Bochs-Emulated Minos



UnderStand



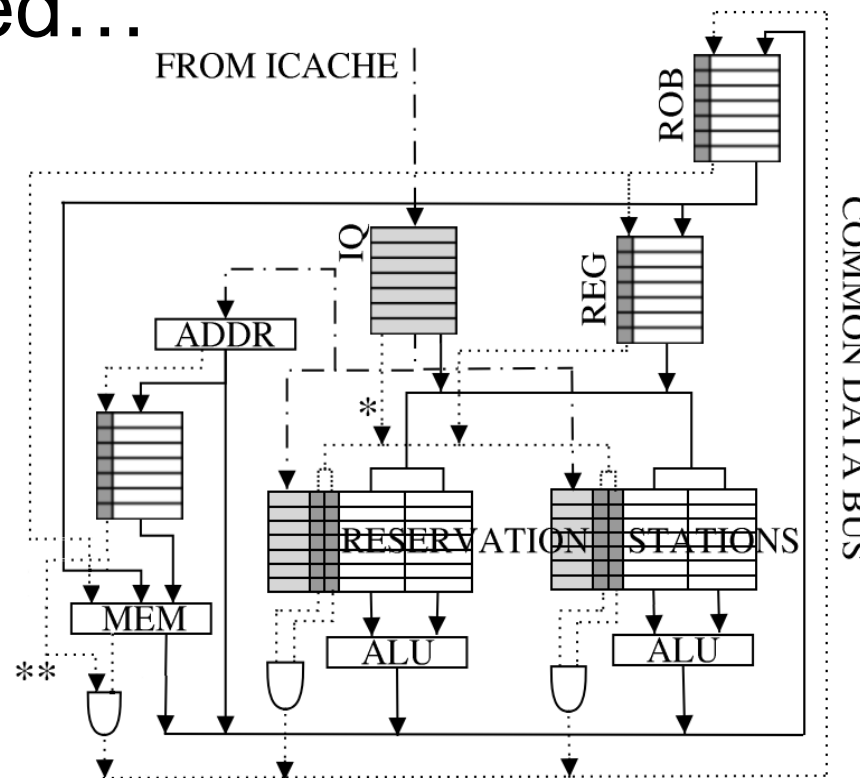
# Minos



- Un-trusted low-integrity information is being propagated, combined, modified...

MINOS

Bochs



jmp  
call  
ret

Tracing  
Without  
Modifying  
OS kernel or  
Applications  
(no source code  
or binary rewrite)

# Biba's Low-water-mark Integrity Policy



- Tracks the “taintedness” of data
- Access controls are based on accesses a subject has made in the past

# When an attack is caught...



- Control-flow related instructions → low integrity data
  - Examples: call, jmp, ret





## FP & FN

- Against real attacks (30~50 per months) under 4 static IP addresses
  - We have changed/reconfigured to try different software patches and versions.
  - Many high-frequency attacks were patched.
- Control Flow Hijack
  - Zero false positive (a proof?)
- How about False Negative?
  - Information flow, and virtual memory



08/29/2005

# MINOS → DACODA



UnderStand

exploit → vulnerability → invariant → signature

# MINOS → DACODA



UnderStand



- Buttercup: NOMS'2004
- Minos: Micro'2004
- Minos Honey-pot: DIMVA'2005



# MINOS → DACODA



UnderStand

- Buttercup: NOMS'2004
- Minos: Micro'2004
- Minos Honey-pot: DIMVA'2005
- **DACODA: ACM CCS'2005**
  - UnderStand the amount of possible Polymorphisms
  - “negative results”
    - Why Buttercup, Taintcheck, Polygraph, Earlybird, and basically all existing network-based solutions won't work!!

# Understanding the Exploits



UnderStand

- Attacks are much more complex than our naïve belief, based on our analysis of 14+ recent exploits

Table 1: Exploits Analyzed by DACODA

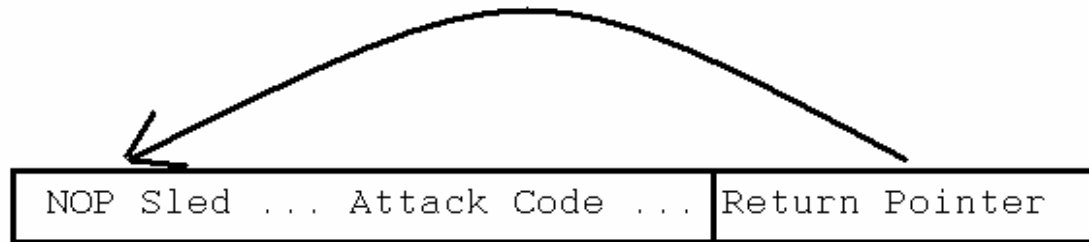
Exploit	Operating System	Class	Ports
LSASS (Sasser)	Windows XP	Buffer Overflow	445 TCP
DCOM RPC (Blaster)	Windows XP	Buffer Overflow	135 TCP
LSASS ASN.1	Windows XP	Double free()	445 TCP
Unidentified (RPCSS?)	Windows Whistler	Buffer Overflow	135 TCP
SQL Name Resolution (Slammer)	Windows Whistler	Buffer Overflow	1434 UDP
SQL Authentication (Hello)	Windows Whistler	Buffer Overflow	1433 TCP
IIS (Code Red II)	Windows Whistler	Buffer Overflow	80 TCP
wu-ftp Format String	Red Hat Linux 6.2	Format String	21 TCP
wu-ftp Heap Globbing	Red Hat Linux 6.2	Double free()	80 TCP
rpc.statd	Red Hat Linux 6.2	Buffer Overflow	?? and ??
innd	Red Hat Linux 6.2	Buffer Overflow	119 TCP
Apache Chunk Handling	OpenBSD 3.1	Buffer Overflow	80 TCP
ntpd	FreeBSD 4.2	Buffer Overflow	?? TCP
Turkey	FreeBSD 4.2	Buffer Overflow	21 TCP

Table 3: Processes Involved

Exploit Name	CR3 Values	Processes
LSASS (Sasser)	0x00039000, 0x03fef000	SYSTEM and lsass.exe
DCOM RPC (Blaster)	0x00039000, 0x04572000	SYSTEM and svchost.exe
LSASS ASN.1	0x00039000, 0x044e2000, 0x04ed5000, 0x01877000	SYSTEM, LSASS, ??, and ??
Unidentified (RPCSS?)	0x07346000	??
SQL Name Resolution (Slammer)	0x075b6000??	SQL Server
SQL Authentication (Hello)	0x00039000, 0x07ba0000	SYSTEM and SQL Server
IIS (Code Red II)	0x00039000, 0x07ece000	SYSTEM and IIS web server
wu-ftp Format String	0x07dc4000, 0x04c76000	wu-ftp and ???
wu-ftp Heap Globbing	??	??
rpc.statd	0x07de4000, 0x07b83000	portmap and rpc.statd
inn	0x65cc000, 0x067a1000	inn and nnrpd
Apache Chunk Handling	0x02ee9000	httpd
ntpd	??	??
Turkey	??	??

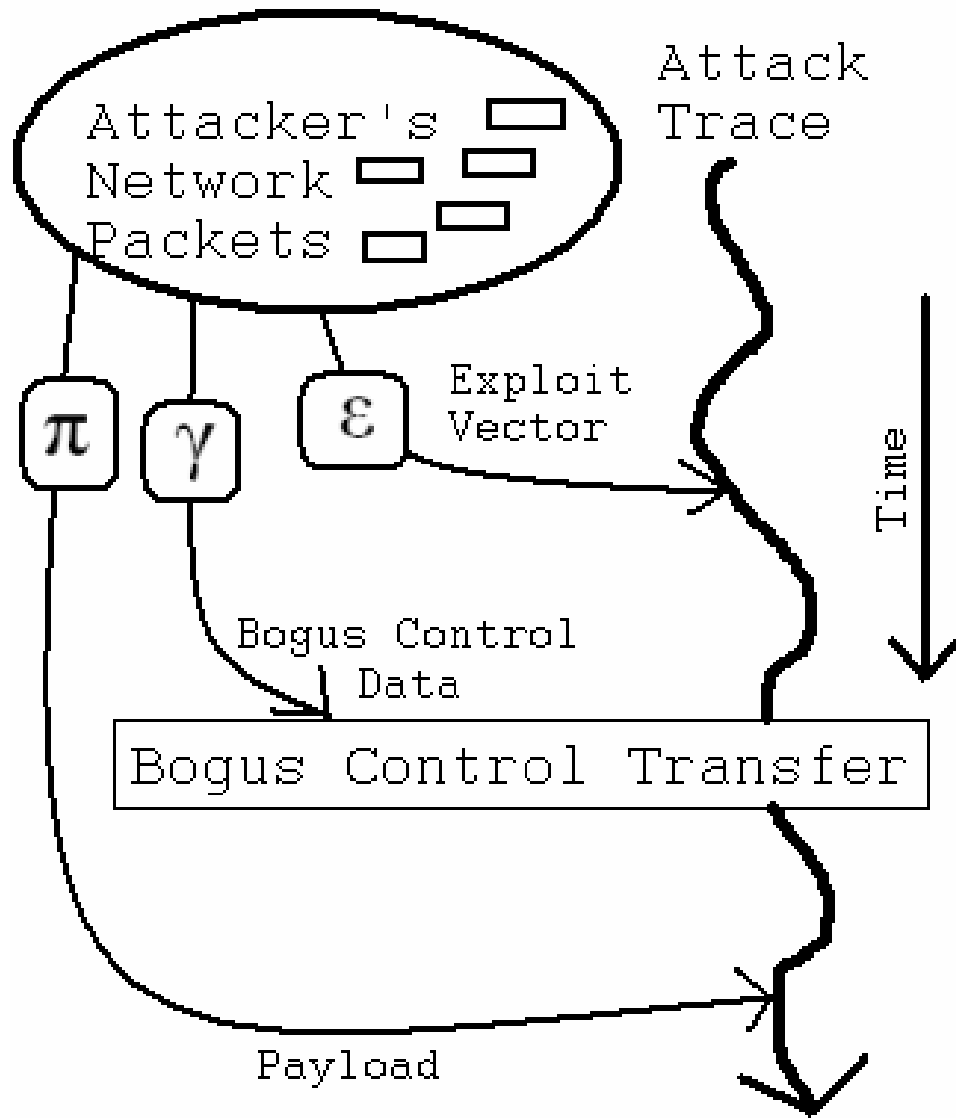


# Simple view of buffer overflows

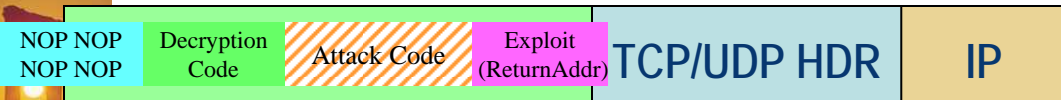




UnderStand



How can each of the stages be polymorphic?



System State Changes

# Register Spring



UnderStand

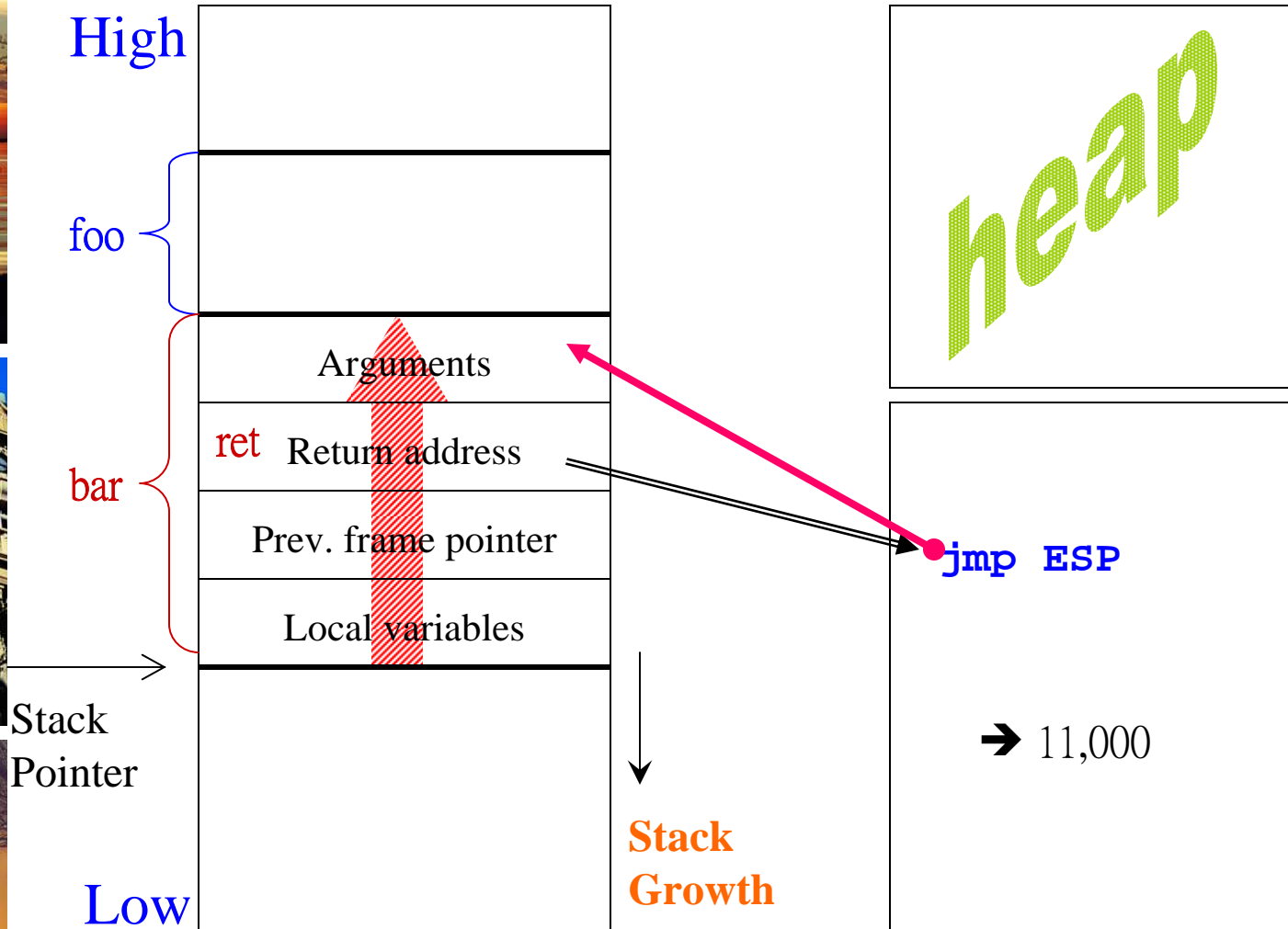
Exploit Name	Superfluous Bytes	First Hop	Interesting Coding Techniques
SQL Hello	>500	Register Spring	Self-modifying code
Slammer Worm	>90	Register Spring	Code is also packet buffer
Code Red II	>200	Register Spring	Various
RPC DCOM	>150	Register Spring	Self-modifying code
LSASS	>27000	Register Spring	Self-modifying code
ASN.1	>47500	Register Spring	First Level Encoding
wu-ftpd	>380	Directly to Payload	x86 misalignment
ssh	>85000	Large NOP sled	None

**We in general don't know which "thread stack" will be used?! 4 millions in memory differences.**

# Register Spring



UnderStand





## Other registers

- Register springs off of other registers utilize the compiler conventions for managing buffers (i.e. EBX is the “base” register for indexing the base of a buffer, ESI is the “source” register for string operations, EDI is the “destination”, ...)
- Blaster RPC DCOM used EBX, ASN.1 uses EDI, Code Red II used EBX

## DCOM Exploits in svchost (Blaster)



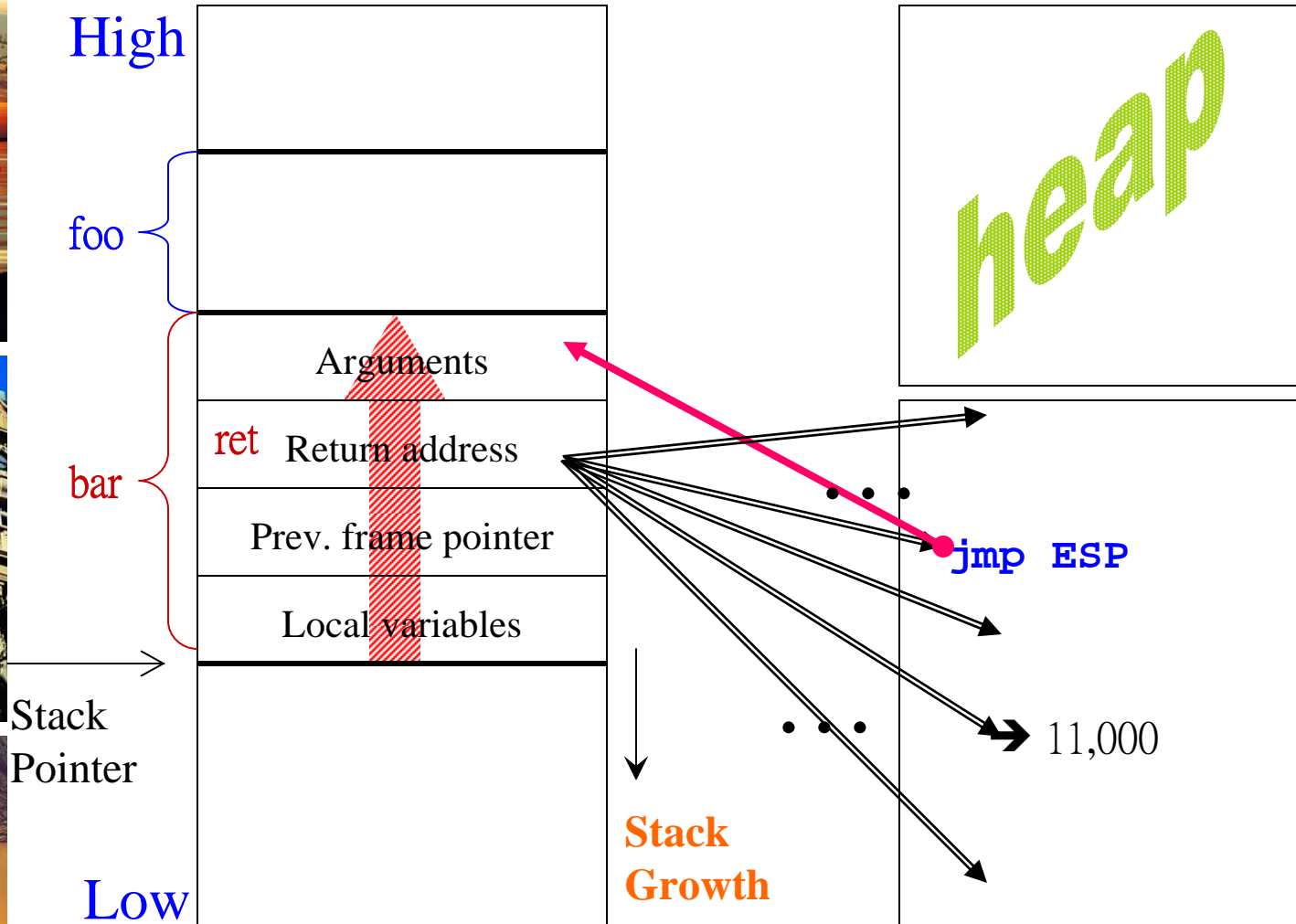
UnderStand

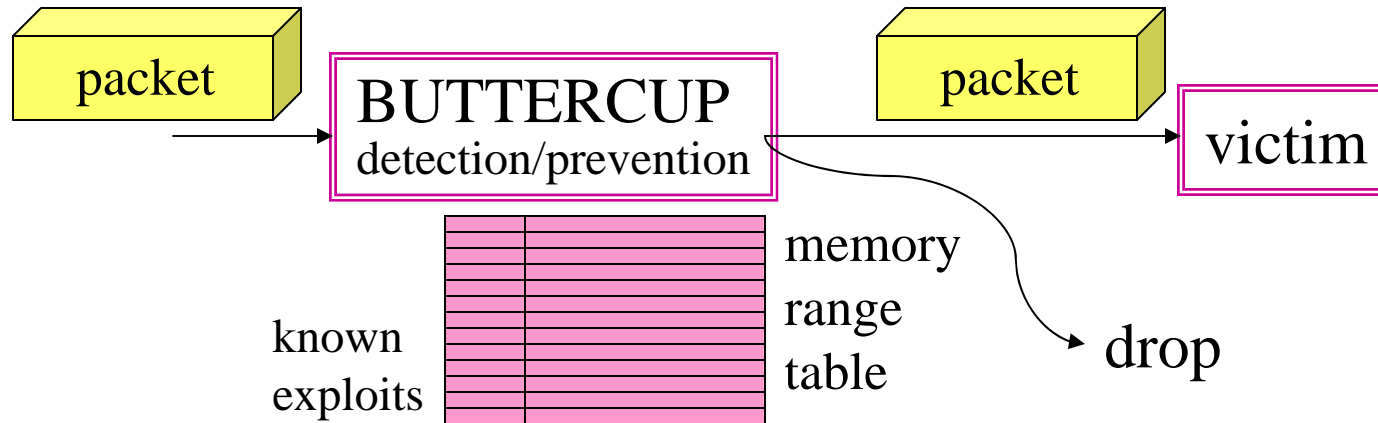
- 0xff 0xd3 is CALL EBX which is the one Blaster used, but JMP EBX (0xff 0xe3) works just as well.
- a little over 11,000 in svchost
- 0x0100139d is the only one that Blaster used and is the one the publicly available DCOM exploit uses.

# Register Spring



UnderStand



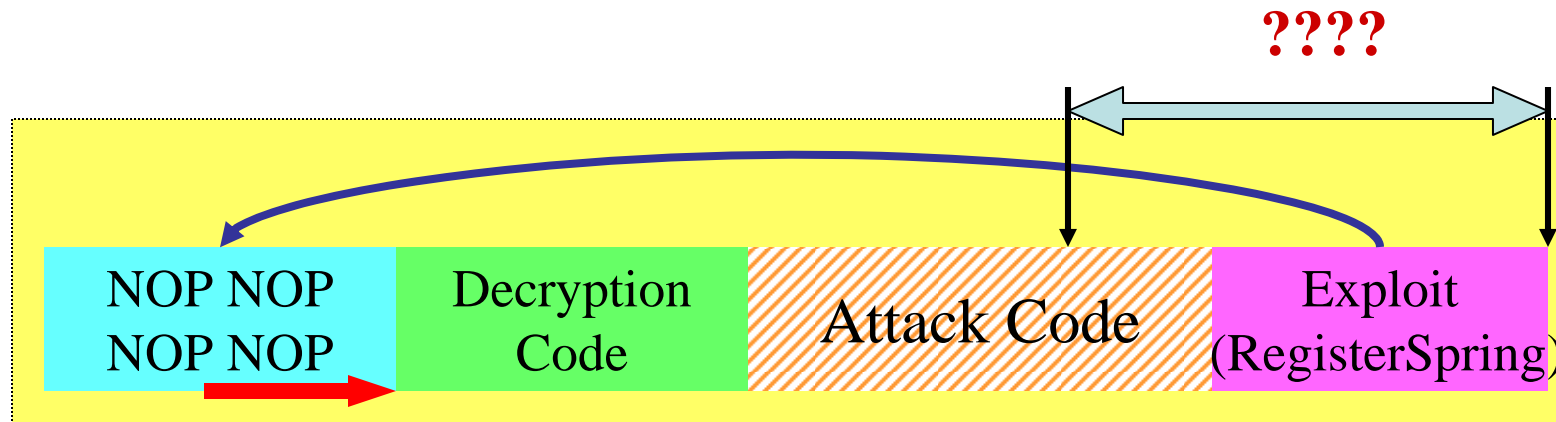


**11,000 Signatures for ONE vulnerability!!**

**False Positive on BUTTERCUP???**



# Register Spring+Polymorphic



~~"0x0100139d"~~





UnderStand

# EBX-based Buttercup

- Among all the memory address for call ebx (0xff 0xd3 -- 11000+ of them), only four of them are around 0x01001\*\*\*, about another 600+ are from 0x719555a4 to 0x71c637b3. But, the rest of them (the majority 10000+) are all from 0x7585149f to 0x77fbc10b.

```

0x0100139d
0x010013a2
0x01001c83
0x01001cc7
0x719555a4 → 0x71c637b3
0x7585149f → 0x77fbc10b
    
```

## Remarks



UnderStand

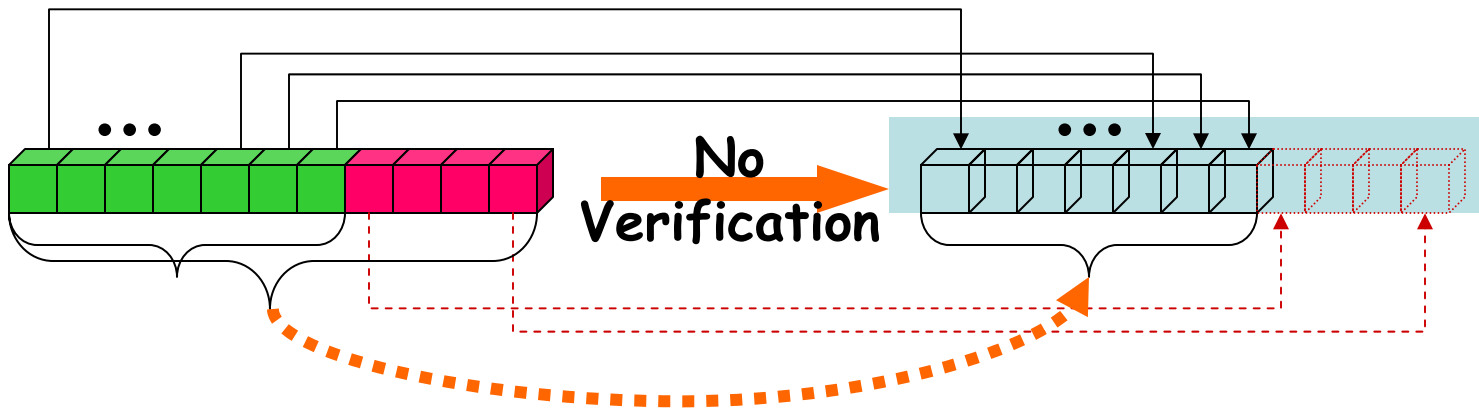
- Software Vulnerability is a very difficult issue to manage, especially **on the wire**.
  - Naïve payload analysis will be much less meaningful
  - Minos is an excellent “host-based” solution
  - Not focus on the intention of the attacker
  - Focus on how their code can get in!
- Minos + DACODA → UnderStand
  - But, NO simple & yet powerful “signature”!



# Vulnerability → Primitive

- Primitive

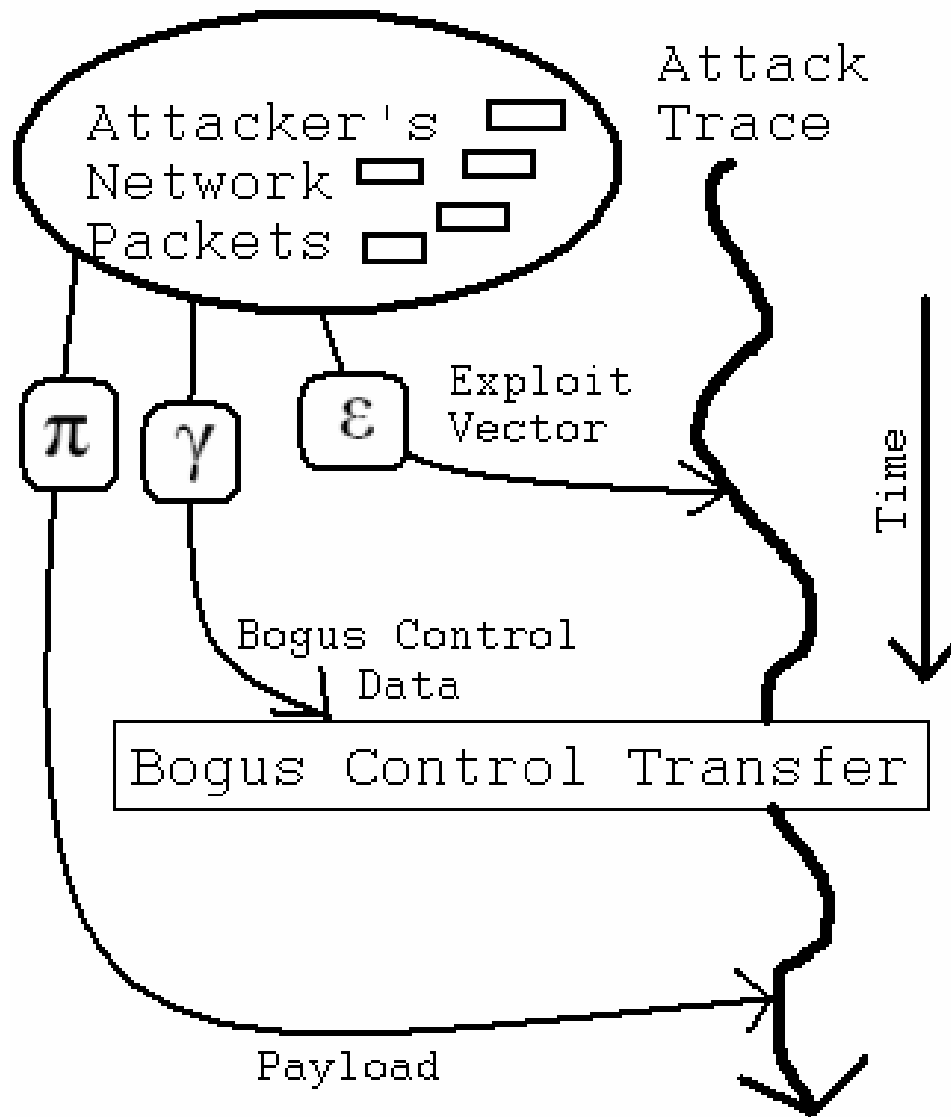
- The capability for the attacker to put a value in a particular memory address.
- A memory system state change



And, we "might" have to perform such analysis on the wire!!

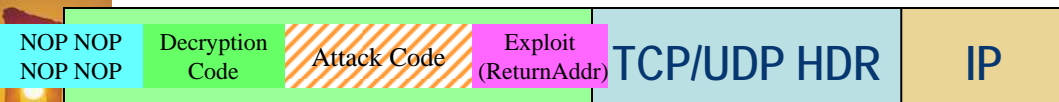


UnderStand



Focus on “Primitives” being used in the “Epsilon” phase!

→ Application dependent analysis

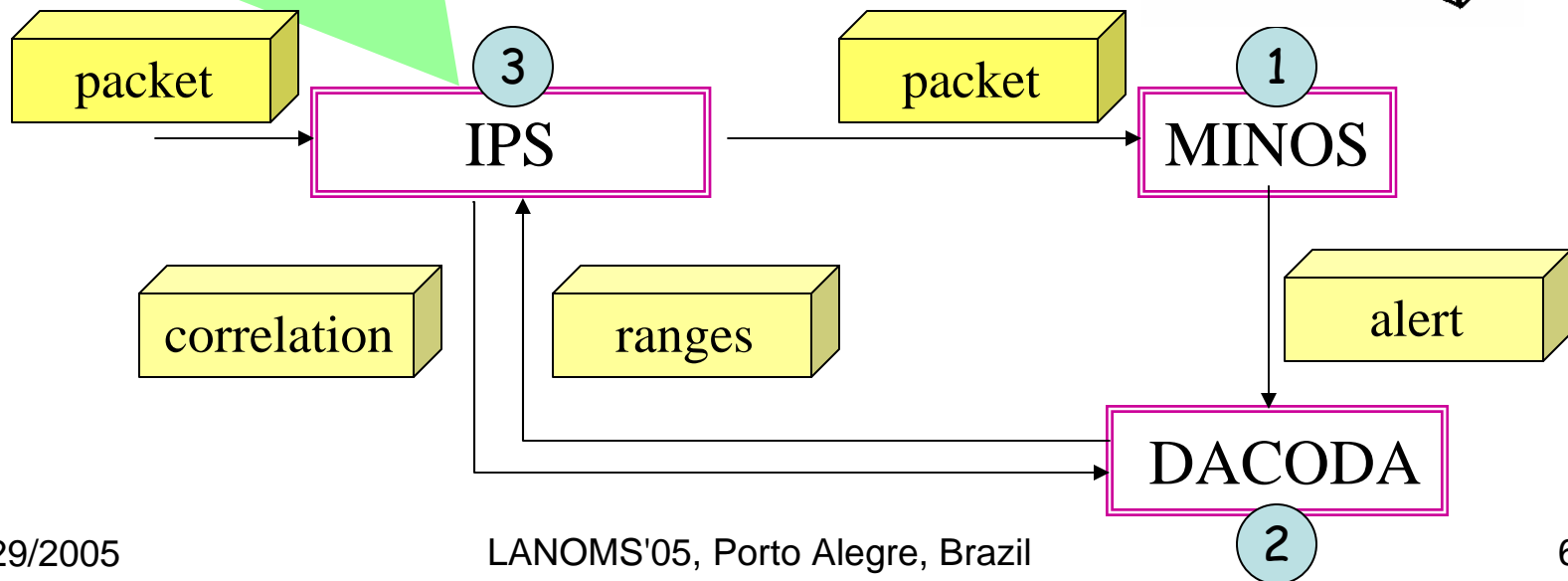
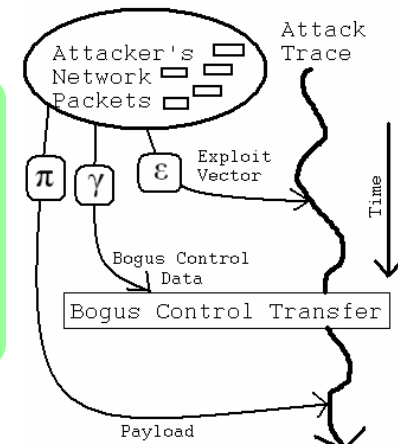
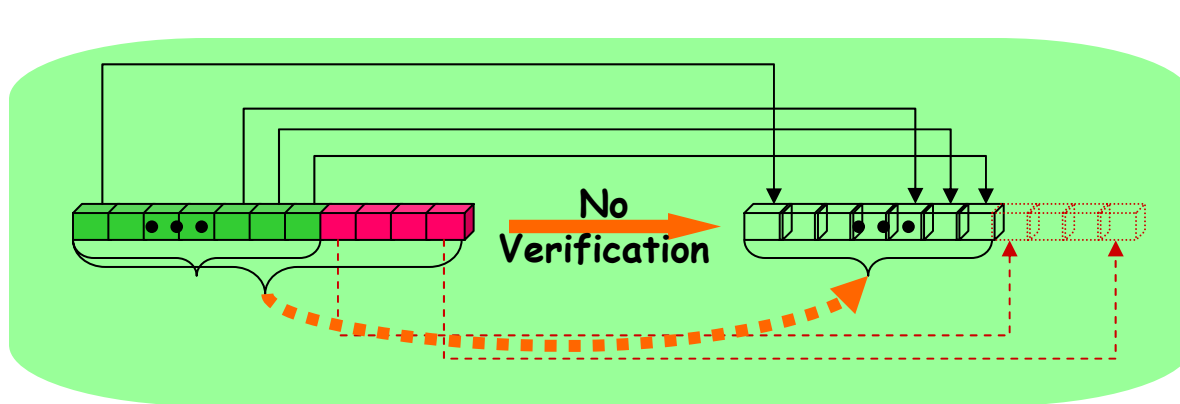


System State Changes

# Unknown-vulnerability Collaborative Defense



UnderStand





UnderStand

# Acknowledgements

- Faculty members:
  - Fred Chong, Zhendong Su, Karl Levitt
- PhD students
  - Jed Crandall, Jason Coit, Gary Hong, Daniela Alvim Seabra de Oliveira
- Thank LANOMS'2005!
- [wu@cs.ucdavis.edu](mailto:wu@cs.ucdavis.edu)



UnderStand



UnderStand

If the programmer of foo.DLL did the following:

X\_low:           low integrity data  
Y\_high:           high integrity data

```
(1).  if (X_low == 5) Y_high = 5;  
      Y_high = X_low = 5;  
(2).  Y_high = Z_table[X_low];  
(3).  Y_high = 0;  
      while (X_low--) Y_high++;
```

**Not a very common programming style though!**