e-NeXSh: OS Fortification

Protecting Software from Internet Malware

Gaurav S. Kc, Angelos D. Keromytis

Columbia University

Bane of the Internet

- Internet Malware
  - Internet worms and Internet-cracking tools
  - Override program control to execute malcode
- Internet Worms
  - Morris '88, Code Red II '01, Nimda '01, Slapper '02, Blaster '03, MS-SQL Slammer '03, Sasser '04
  - Automatic propagation
- Internet Crackers
  - "**j00** *got* **h4x0r3d**!!"
- After breaking in, malware will:
  - Create backdoors, install rootkits (conceal malcode existence), join a bot-net, generate spam
- e-NeXSh can thwart such malware

Worms, viruses prove costly

The estimated cleanup and lost productivity costs of worms and viruses add up:

Year	Virus/worm	Estimated damage
1999	Melissa virus	$80 million
2000	Love Bug virus	$10 billion
2001	Code Red I and II worms	$2.6 billion
2001	Nimda virus	$590 million to $2 billion
2002	Klez worm	$9 billion
2003	Slammer worm	$1 billion

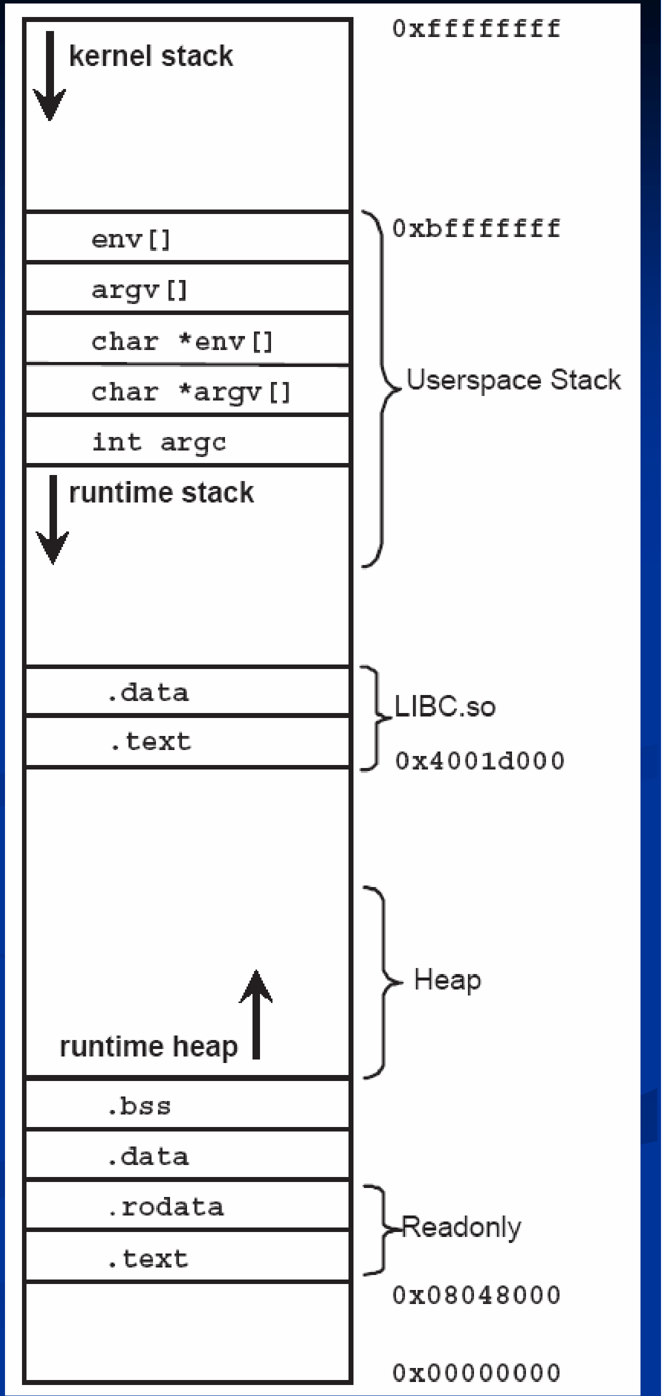
Source: USA TODAY research

Outline

- Software Run-Time Environments (x86/Linux)
  - Bugs, and Breaches: Anatomy of Attacks
- e-NeXSh: OS Fortification
- Related Work
- Conclusions

Process Run-Time

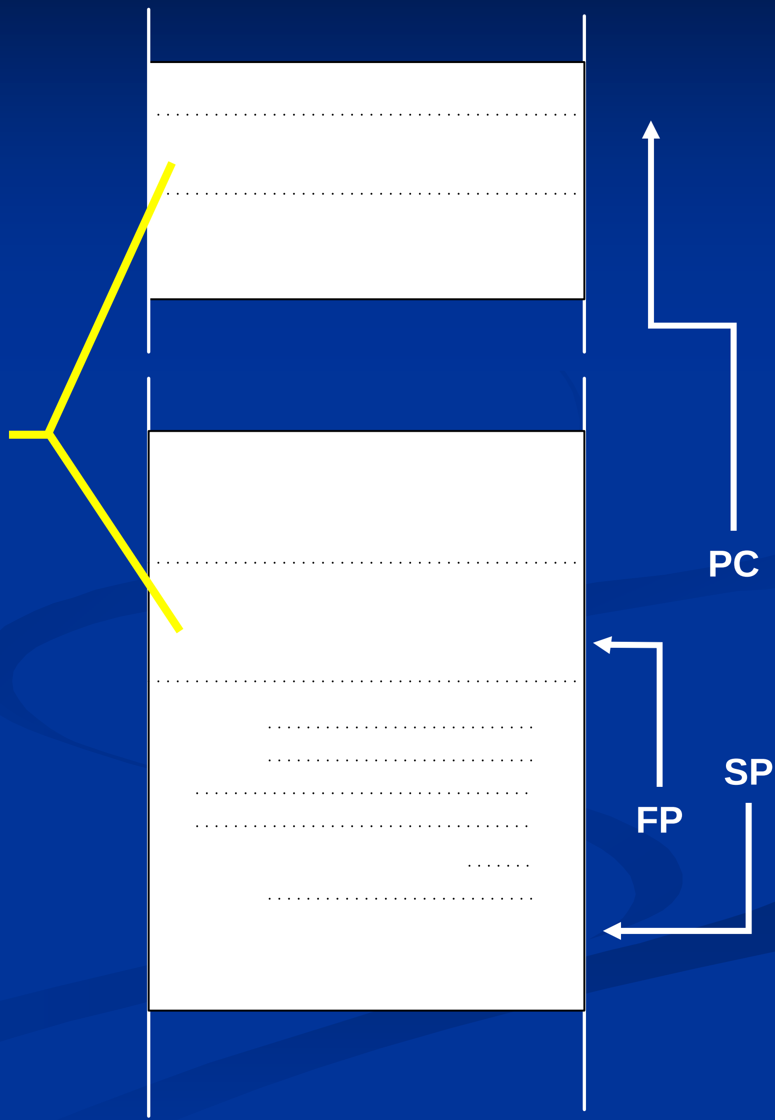
- Linux: Multi-processor OS
  - Resource manager and scheduler
    - Inter-process communication (IPC)
    - Access: network, persistent storage devices
    - Process scheduling and context-switching
- Process: abstraction of program in execution
  - 4GB of virtual memory
  - Code + data segments
  - **`.stack`** segment
    - Activation records



Process Run-Time

- Activation records

Activation Frame Header
return_address, old_frame_pointer




```
void function(char *s, float y, int x) {
  int a;
  int b;
  char buffer[SIZE];
  int c;
  strcpy(buffer, s);
  return;
}
```

Invoking System Calls

- Applications access kernel resources

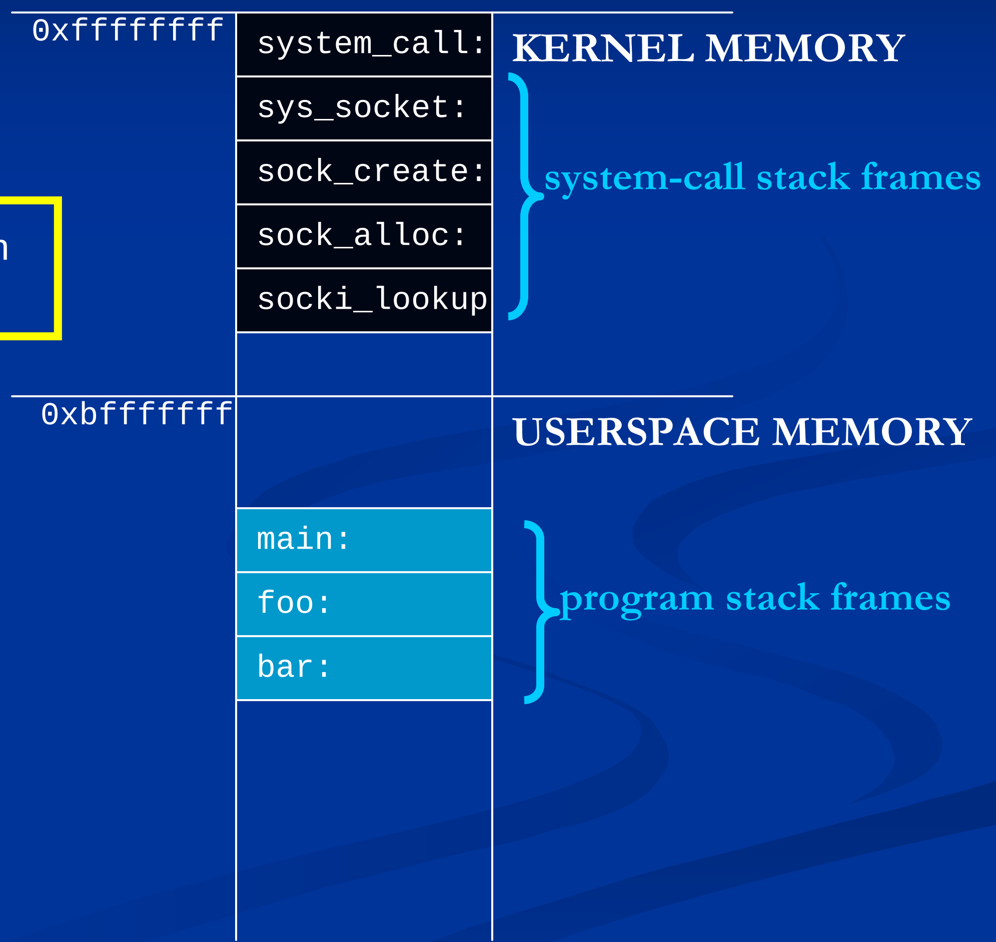
Machine instruction in `.text` section

program.c

```
bar() {
  ...
  int $0x80   ; trap instr.
  ...
}

foo()  { bar(); }
main() { foo(); }
```

kernel

```
system_call()  { call *0x0(,%eax,4); }
sys_socket()   { sock_create();      }
sock_create()  { sock_alloc();       }
sock_alloc()   { socki_lookup();     }
socki_lookup() { ...                 }
```



System Calls via LIBC

program.c

```
bar() {
  socket(...);
}

foo()  { bar(); }
main() { foo(); }
```

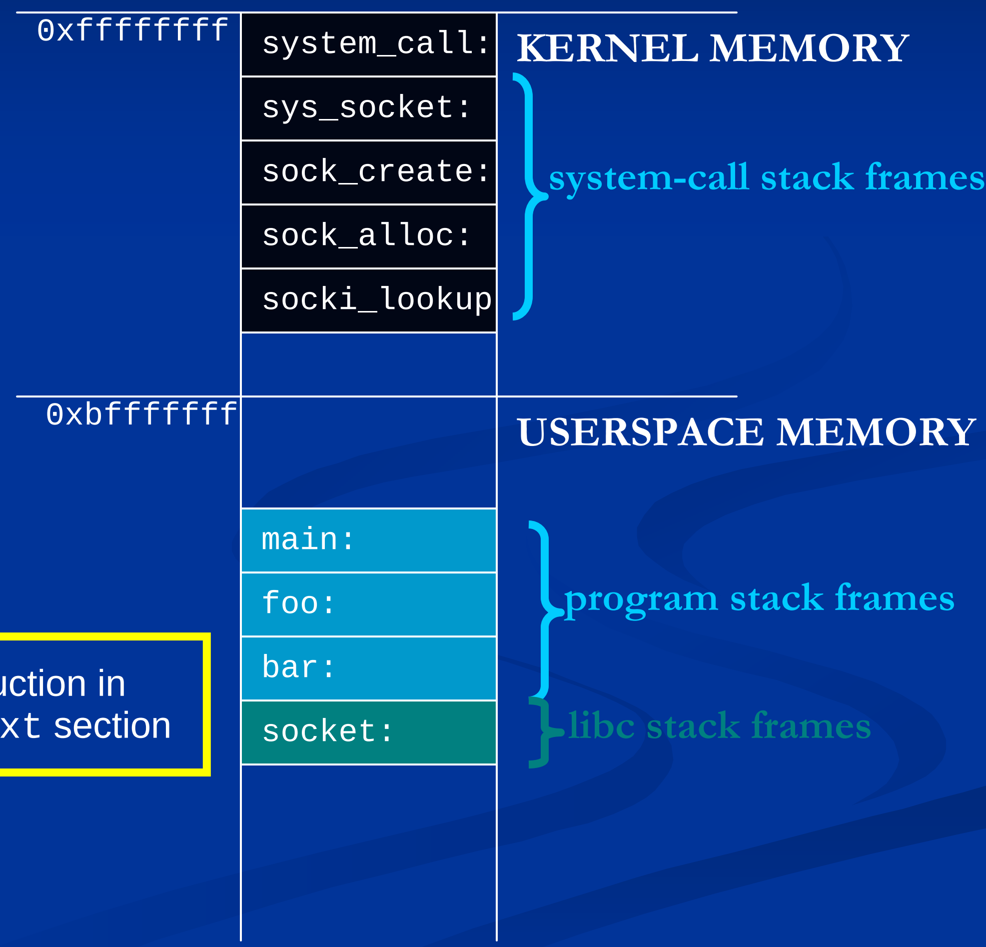
libc.so

```
socket() {
  ...
  int $0x80  ; trap instr.
  ...
}
```

Machine instruction in LIBC `.text` section

kernel

```
system_call()  { call *0x0(,%eax,4); }
sys_socket()   { sock_create();      }
sock_create()  { sock_alloc();       }
sock_alloc()   { socki_lookup();     }
socki_lookup() { ...                 }
```



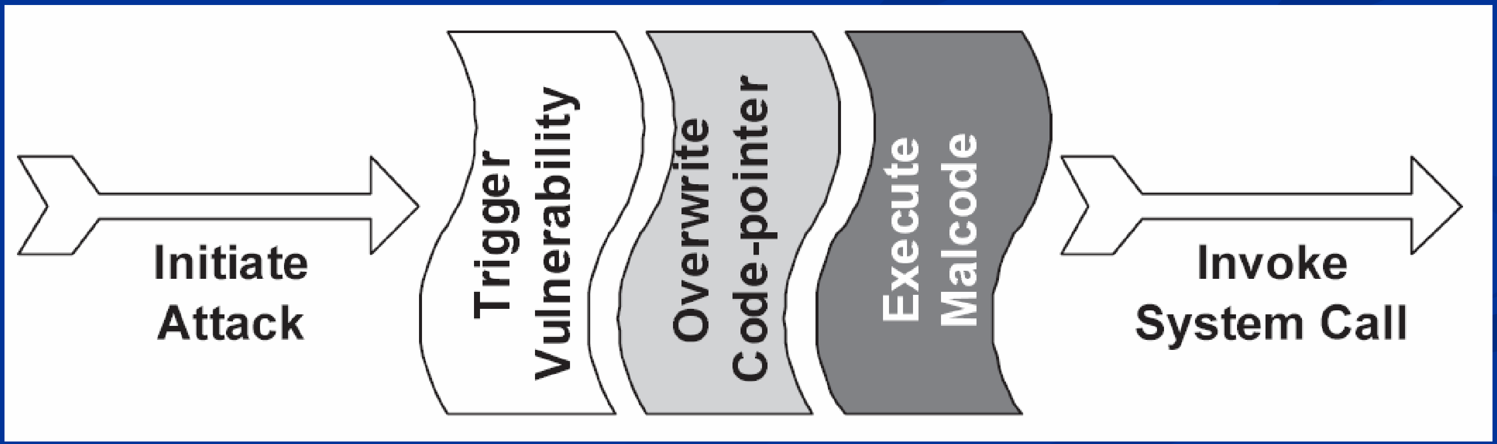
Security Vulnerabilities

- C: A low-level, systems language with unsafe features
  - No bounds-checking. Not strongly typed.
    - Arbitrary memory overwrites
- Common security vulnerabilities
  - Buffer overflows
  - Format-string vulnerability
  - Integer overflows
  - Double-free vulnerability

Anatomy of a Process-Subversion Attack

- Analysis of common attack techniques
  - Phrack magazine, BugTraq, worms in "the wild"

- Stages of a process-subversion attack
  1. Trigger vulnerability in software
  2. Overwrite code pointer
  3. Execute malcode of the attacker's choosing, and invoke system calls



Process-Subversion Attacks contd.

- *Component Elements* (C.E.) of an attack
  1. exploitable vulnerability
     e.g., buffer overflows, format-string vulnerabilities
  2. overwritable code pointer
     e.g., return address, function pointer variables
  3. executable malcode
     e.g., machine code injected into data memory, existing application or LIBC code

Focus of e-NeXSh!

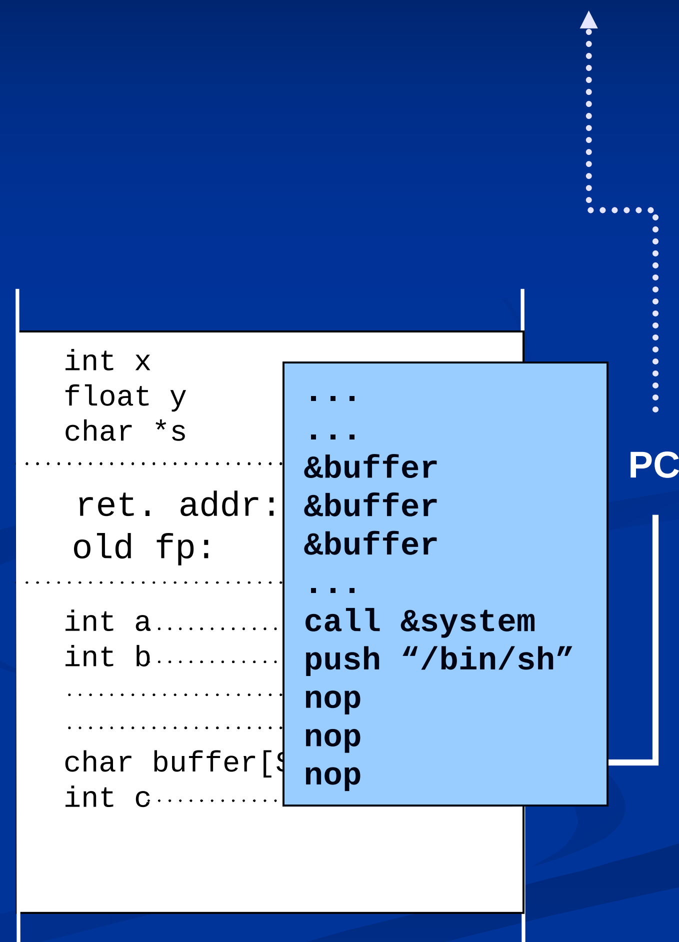
Methods of Attack

```
void function(char *s, float y, int x) {
  int a;
  int b;
  char buffer[SIZE];
  int c;
  strcpy(buffer, s);
  return;
}
```

Buffer-overflow vulnerability

<u>Stacksmashing (LIBC-Based)</u>

Overrun buffer
Overwrite return address
Injected code invokes LIBC function



Outline

- Software Run-Time Environments (x86/Linux)
  - Bugs, and Breaches: Anatomy of Attacks
- e-NeXSh: OS Fortification
- Related Work
- Conclusions

e-NeXSh: Monitoring Processes for Anomalous and Malicious Behaviour

- Monitor LIBC function invocations

```
If (call stack doesn't match call graph)
    exit (LIBC-based attack);
```

- Monitor system-call invocations

```
If (system call invoked from data memory)
    exit (injected code execution);
```

- Explicit policy definitions ~~required!~~
  - Use program disassembly information and memory layout.
- Code can still execute on stack/heap, just cannot invoke system calls directly or via LIBC functions

e-NeXSh: System Calls via LIBC

program.c

```
bar() {
  socket(...);
}

foo()  { bar(); }
main() { foo(); }
```

e-NeXSh.so

```
socket() {
  // validate call stack
  libc.so :: socket();
}
```

Valid call stack

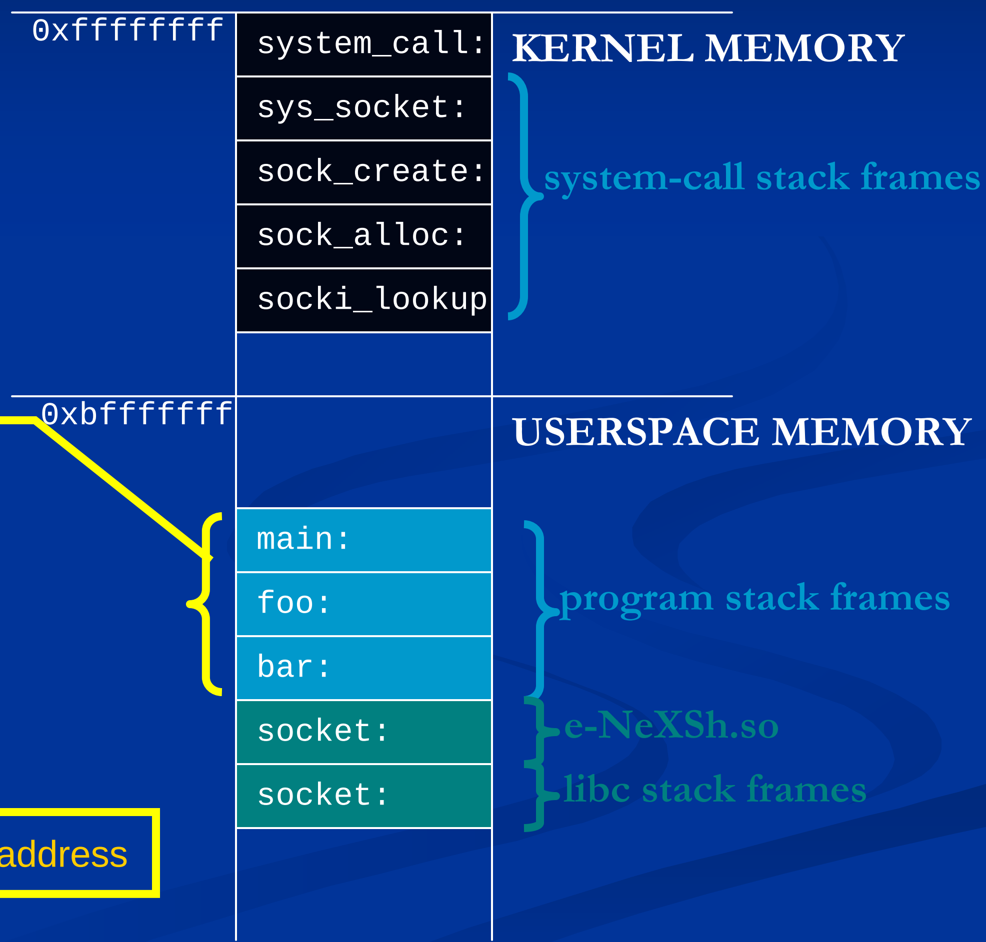
libc.so

```
socket() {
  int $0x80  ; trap instr.
}
```

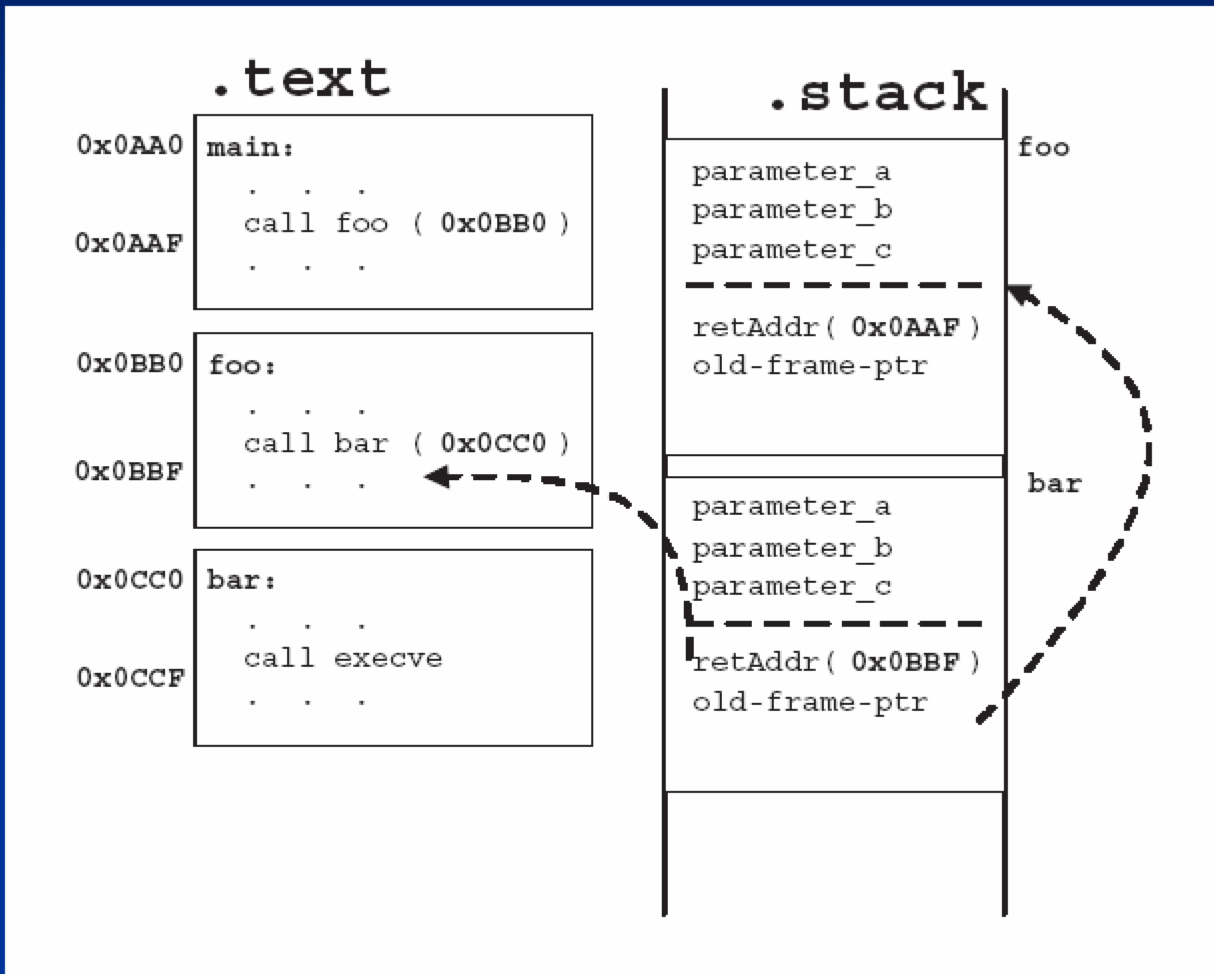
kernel

```
system_call()  {
  // validate "return address"
  call *0x0(,%eax,4);
}
sys_socket()   { sock_create();      }
sock_create()  { sock_alloc();       }
```

Valid return address



e-NeXSh: Validating the Call Stack



e-NeXSh against LIBC attacks

program.c

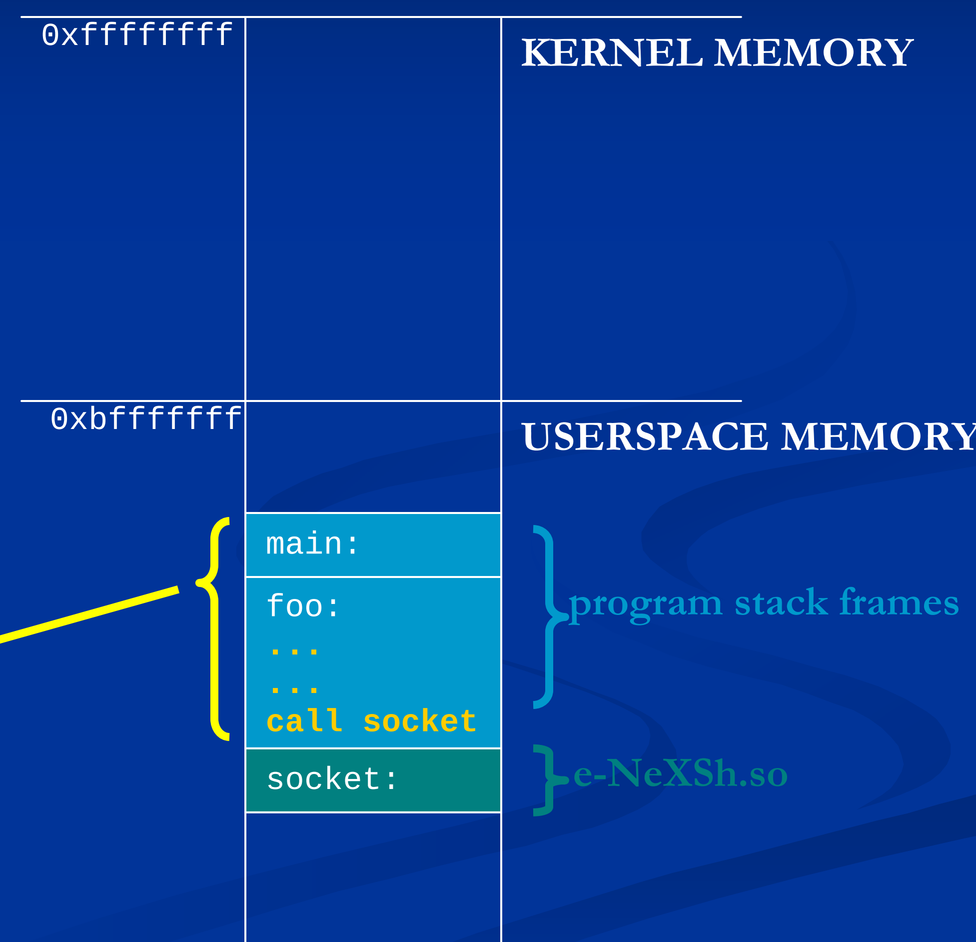
```
bar() {
  socket(...);
}

foo()  { bar(); }
main() { foo(); }
```

e-NeXSh.so

```
socket() {
  // validate call stack
  libc.so :: socket();
}
```

exit(-1)

INVALID call stack



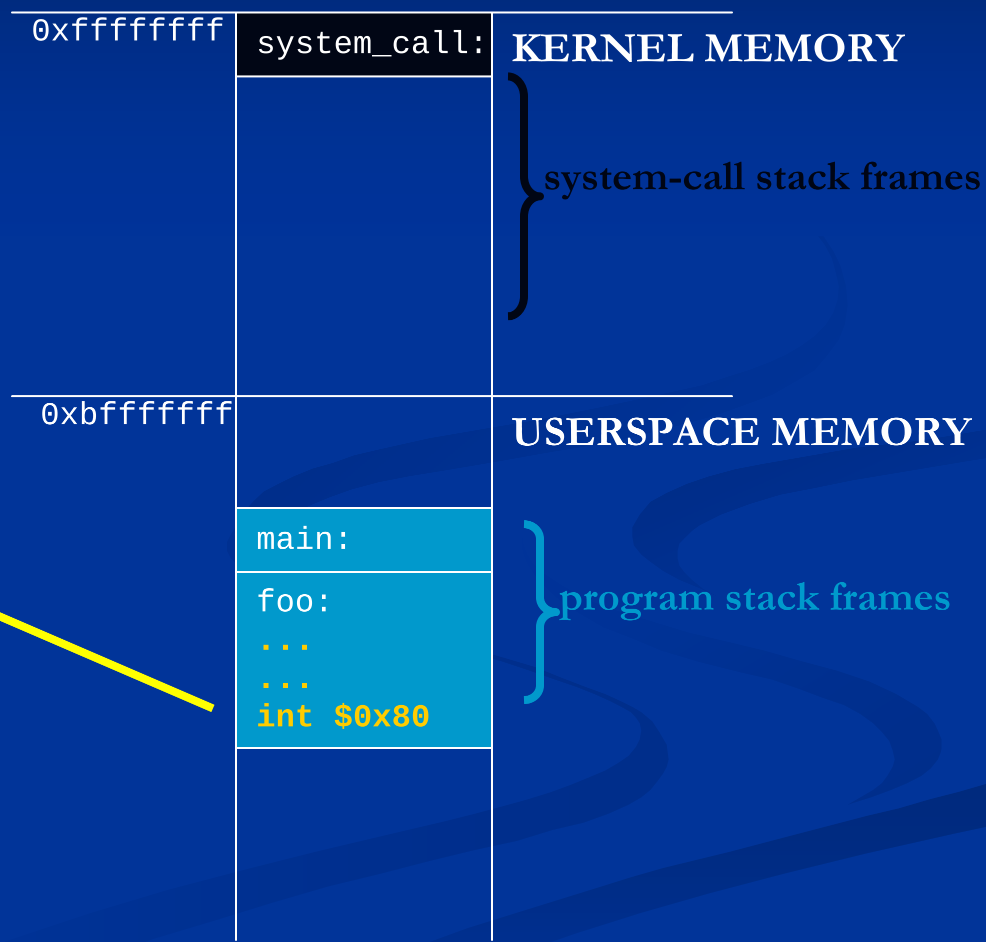
e-NeXSh: User-Space Component

- Interposition of calls to LIBC functions
  - Define `LD_PRELOAD` environment variable
- Validate call stacks
  - Conduct stack walk to determine caller-callee pairs
  - Validate caller-callee pairs against program code
    - Derive function boundaries from disassembly information
    - Inspect `.text` segment to determine `call` instructions where caller invokes callee
  - If okay, allow through to LIBC

e-NeXSh against Injected Code

program.c

```
bar() {
  socket(...);
}

foo()  { bar(); }
main() { foo(); }
```



INVALID return address

kernel

```
system_call()  {
  // validate "return address"
  call *0x0(,%eax,4);
}
sys_socket()   { sock_create();      }
sock_create()  { sock_alloc();       }
```

exit(-1)

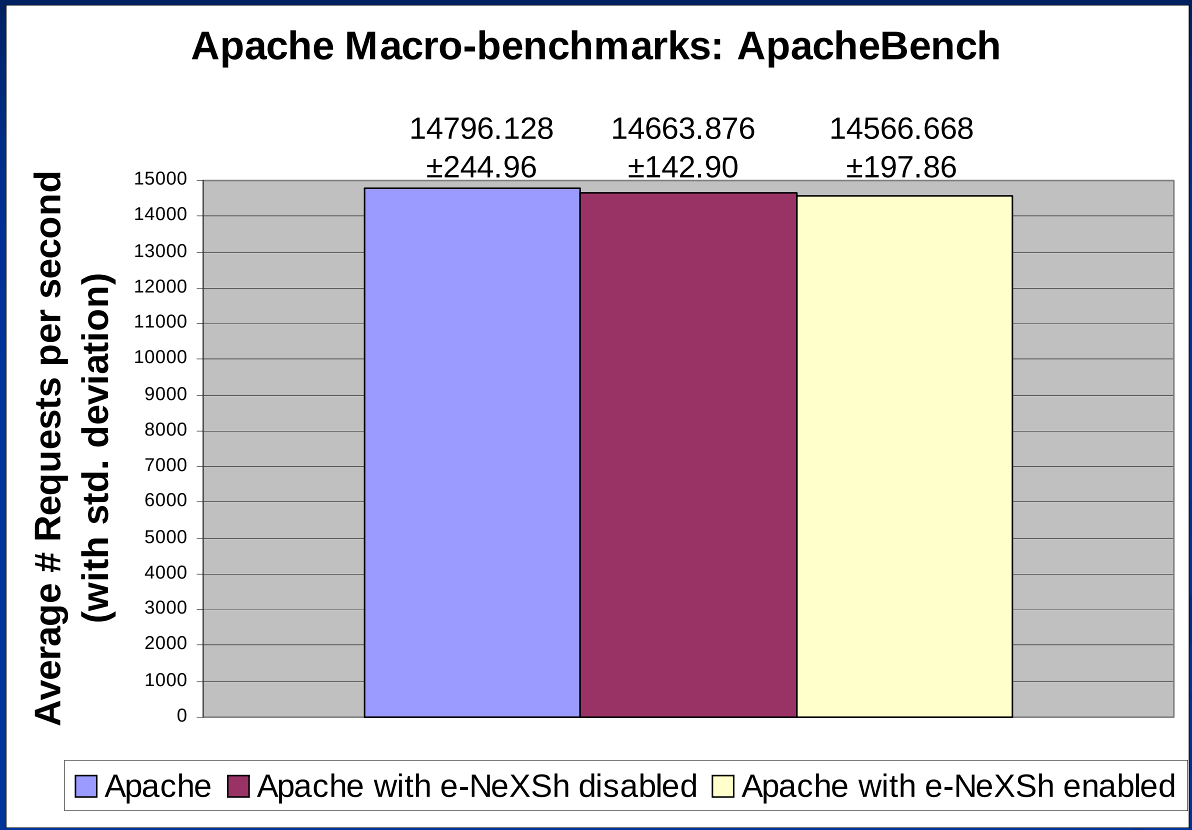
e-NeXSh: Kernel-Mode Component

- Interposition of system calls in kernel
  - Extended the system-call handler code
- Validate call sites of system-call invocations
  - Extract "return address" of system call from stack
  - Match against process' virtual memory address ranges for all `.text` segments
    - `int $0x80` instruction must exist in a `.text` segment
  - If okay, allow through to system call function

e-NeXSh: faq

- Can the attacker change write-permissions on data pages?
  - No, this can only be done via a system call
- Can the attacker spoof the return address for system-call invocations?
  - No, the kernel's system-call handler sets this up
- Can the attacker fake a valid stack, and then invoke LIBC?
  - No, we can randomise the offsets for the .stack and .text segments, and also randomise the old-FP and return addresses on the stack. This prevents an attacker from creating a seemingly valid, but fake stack.
- What are the modifications to Linux?
  - Very minimal: assembly code (~10LOC) and C code (~50LOC) in the kernel. ~100LOC of C code for LIBC wrappers
- What are the performance overheads?
  - See results for ApacheBench benchmarks and UNIX utilities

Performance Overhead



1.55% average decrease (± 2.14% std. deviation) in request-handling capacity for Apache-1.3.23-11

Performance Overhead

Benchmark	Normal in seconds	e-NeXSh in seconds	Overhead in percent
`ctags`	9.98±0.14	9.91±0.10	-0.60±1.93
`gzip`	10.98±0.62	11.19±0.44	2.09±6.45
`scp`	6.30±0.04	6.29±0.04	-0.15±0.96
`tar`	12.89±0.28	13.12±0.46	1.84±3.91

- e-NeXSh macro-benchmark: UNIX utilities
  - Processing **`glibc-2.2.5`**
    - **`ctags -R`**; **`tar -c`**; **`gzip`**; **`scp user@localhost:`**
  - Larger standard deviation than (at times, negative) overheads

Limitations. Future Work

- Indirect call instructions in stack trace?
  - Harder to validate call stack
  - Need list of valid indirect callers for functions in call stack
    - Static data-flow analysis to determine all run-time values for function pointers, C++ VPTRs
    - Collect training data to determine valid call stacks with indirect calls

Outline

- Software Run-Time Environments (x86/Linux)
  - Bugs and Breaches: Anatomy of Attacks
- e-NeXSh: OS Fortification
- Related Work
  - System-call interposition
  - Preventing execution of injected code
  - LIBC address-space obfuscation
- Conclusions

Related Work: System-Call Interposition

- Host-based Intrusion Detection Systems (IDS)
  - Forrest (HotOS-97), Wagner (S&P-01)
  - Co-relate observed <u>sequences</u> of system calls with static FSM-based models to detect intrusions
  - Imprecise (false positives) or high overheads
  - Vulnerable to mimicry attacks, Wagner (CCS-02)

Related Work: Non-Executable Stack/Heap

- Instruction-Set Randomisation
  - Barrantes (CCS-03), Kc (CCS-03)
  - Randomised machine instruction sets to disable injected code
  - High overhead due to software emulation of processor
- Non-Executable Stack/Heap
  - Openwall, PaX, OpenBSD W^X, Redhat ExecShield, Intel NX
  - Disable execution of injected code in data memory
  - Complex workarounds required for applications with a genuine need for an executable stack or heap

Related Work: Address-Space Randomisation

- Obfuscation of LIBC Functions' Addresses
  - Bhatkar (SEC-03), Chew (CMU-TR-02), PaX-ASLR
  - Prevent use of LIBC functions in attack
  - Vulnerable to brute-forcing, Shacham (CCS-04)

Conclusions

- e-NeXSh is a simple, low overhead OS-fortification technique.
  - Implemented prototype on the Linux kernel
  - Thwarts malicious invocations of system calls, both directly by injected code, and via LIBC functions