Secure Bit: Buffer Overflow Protection

"Give me a (little) bit, and I will solve buffer overflow."

Kerkr Piromsopa, Ph.D.
Department of Computer Engineering

Overview

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

Simple Buffer Overflow

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    int age;
    char name[8];
    char tmp[20];
    printf("Enter your age:");
    gets(tmp);
    age = atoi(tmp);
    printf("Enter your name:");
    gets(name);
    printf("-----------
RichardJ is %d years old
", name, age);
    return 0;
}
```

What's wrong?

Stack Buffer Overflows at Work

- Exception handlers
- Function pointers
- Virtual methods
- Return-address attacks (Stack smashing)
- Frame-pointer attacks

Michael Howard, Microsoft.
Secure Bit: Buffer-Overflow Protection

Department of Computer Engineering, Chulalongkorn University

Sample Buffer-Overflow Attack

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

```c
int vulnerable(char **argv) {
    int x;
    char *ptr;
    char buffer[30];
    ptr = buffer;
    printf("ptr\%p - before\n", ptr);
    strcpy(ptr, argv[1]);
    printf("ptr\%p - after\n", ptr);
    strcpy(ptr, argv[2]);
    printf("done\n");
}
```

Jump Slot
Function Pointer (Control Data)

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

Classification of Buffer Overflow Protection

- Buffer-overflow Protection
- Static Analysis
- Dynamic Analysis
- Address Protection
- Input Protection
- Bounds Checking
- Obfuscation
- Isolation
- Non-executable
- Sandbox
- Input Protection
- Semantic Analysis
- lexical Analysis
- Address Protection
- Isolation

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
- Address Encode
- Copy of Address

- Tags

Other Dynamic Solutions

- Bound Checking
- Obfuscation

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

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

Input Protection

- Input must not be used as control data
- Boundary
- 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

Untrustworthy domains

- Buffer
Isolation

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

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

SimpleScalar

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

Split Stack

- Separate Control and Data Stack

By UIUC


**IBM ProPolice**

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

**Others (software)**

- Address Obfuscation: (By Stony Brook U., NY.)
  - Randomize the base address of the memory segment
  - Permute the order of variables/structures
  - 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 process key

**StackGuard**

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

**PointGuard**

- Encrypt the pointer for storing, decrypt for dereferencing
- Compatibility?
- Initialization?
- Performance?
- Encryption Algorithm?
Array Bounds Checking/Segmentation

- Symbol table/ Segment Descriptor Table
- Explicitly declare and refer every buffer with base and boundary (including integer, float... Why?)
- Example: Intel IA-32, I-432
- More than 30 times slowdown

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

Address Obfuscation

- 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)
- Stack
  - Heap
  - BSS
  - Data
  - Text (Read-only)

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?

Others (software)

- StackGhost:
  - Use register window
- Split Stack:
  - Separate control and data stack
- SRAS:
  - Use RAS as a validation copy the address
- Overflow?, Speculative update (non-LIFO)?
- RAD:
  - Use mprotect to protect Return Address Repository (RAR)
  - MiraZone RAS, Read-only RAR
  - Performance?
- StackShield:
  - Save redundant copy of return address
  - Compare the return address with the redundant copy
  - Force the code to be in text section
- Legal use of executing code in heap: LISP, OOP

Secure Bit: Buffer-Overflow Protection

January 25, 2006
Department of Computer Engineering, Chulalongkorn University
Secure Bit: Buffer-Overflow Protection

January 25, 2006

Hardware:
Non-Executable Stack/Memory

- Software/Hardware "NX" (currently in the news)
- Heap-based attacks
- Legal use of executable stack?
- Attacks that do not injecting the malicious code/data?

Instruction Set Randomization

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

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
- Performance
- Compatibility and Transparency (e.g. non-LIFO control flows)
- Deployment and Cost

By Columbia U. & Drexel U.
**Compatibility:**

### Non-LIFO Control Flow

Near Entry:
- POP AX; POP instruction pointer (IP)
- PUSH CS; PUSH CS
- POP AX; POP off stack

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

### FAR & NEAR Call Optimization

- FAR & NEAR Call Optimization
- FAR ENTRY:
  - RETF; POP IP and CS off stack
- NEAR ENTRY:
  - POP AX; POP IP into accumulator (AX)
  - POP CS; PUSH CS

**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.**
- **No protection mechanism is perfect, but the reimplementation of all code: BIOS, Kernel, Library (Static & Dynamic), Drivers, applications, etc...**
- **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.
The Theory section discusses the conditions necessary for preventing buffer-overflow attacks. It introduces the concept of preserving the integrity of an address as a sufficient condition for preventing such attacks. The proof follows from the theorem that modifying an address by replacing it using a buffer passed from another domain is a necessary condition for buffer-overflow attacks. The corollary states that if an address cannot be modified (or such modification can be detected), then a buffer-overflow attack is not possible.

The Secure Bit section introduces a protocol for preventing buffer-overflow attacks. Protocol 1 states that passing a buffer across domains always sets the Secure Bit. Hardware enforcement ensures that data from another domain must not be used as a jump target. Similar concepts include the idea that all input is evil until proven otherwise and that data must be validated as it crosses the boundary between untrusted and trusted environments.
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.

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)
If the system cannot enter an unauthorized state and is considered to be a secure system.
QED

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.

Formalization (Cont.)

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 2.
Thus Secure Bit preserves the integrity of the address and is a secure system with respect to buffer-overflow attacks.

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**
Design: Operating System

- Moving data between Kernel and Process in sbt_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

```c
// 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] & sbyte);
    sbyte = sbyte << 1;
}

// 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
    { // set
        vector_s[a20addr_s] |= (sbyte & 0xff);
    }
    else // clear
    { // clear
        vector_s[a20addr_s] &= ~(sbyte & 0xff);
    }
    sbyte = sbyte << 1;
}
```

BOCHS: Memory Interfaces

```c
// 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);
// 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
  - Define SBIT_MACRO (sbit1)
  - Define SBIT.CommandType (sbit1, sbit2)
  - Define SBIT.XOR(sbit1, sbit2)
  - Define SBIT._AND(sbit1, sbit2)
  - Define SBIT.OR(sbit1, sbit2)
  - Define SBIT.NOT(sbit1)

- Set Secure Bit: `sbit = (sbit_mode)`
- Validate Control data:
  - Check if `sbit != 0`
  - `BX_INFO(`call_ew: sbit of target is not secure`);`  
  - `#ifdef HAS_SBIT_EXCEPTION,
    exception(BX_GP_EXCEPTION, 0, 0);`  
  - `#endif`

- About 2410 lines of code in 607 routines affected

### Linux Kernel

- **Threat Surface**
  - Process
  - Etc.
  - Key data to user
  - Standard Input
- **Input Devices**
  - Storage Devices
  - TCP/IP

- **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

### Linux Kernel (Sample Code)

- `__generic_copy_to_user(void *to, const void *from, unsigned long n)
  SET_SBITMODE();
  if (access_ok (VERIFY_WRITE, to, n))
    __copy_user (to, from, n);
  CLR_SBITMODE();
  return n;`
Tested Applications

- gzip (SPEC CPU2000): Lempel-Ziv coding (LZ77) compression algorithm
- 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.

Hacking Test

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

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. Survey of Buffer-Overflow Protection, ACM Computer Survey (submitted)
- Piromsopa, K. and Enbody, R. Buffer-Overflow Protection: The Theory, EIT2006 (submitted)
- Piromsopa, K. and Enbody, R. Arbitrary Copy: Bypassing Buffer-Overflow Protections, EIT2006 (submitted)
- More... (IEEE Micro, WDDD at ISCA)

Demo

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

Without Secure Bit2

With Secure Bit2

Questions?

Thank you

http://www.cp.eng.chula.ac.th/~krerk/sbit2/