Real-World Buffer Overflow Protection in User & Kernel Space

Michael Dalton, Hari Kannan, Christos Kozyrakis

Computer Systems Laboratory
Stanford University
http://raksha.stanford.edu
Motivation

- Buffer overflows remain a critical security threat

- Deployed solutions are insufficient
  - Provide limited protection (NX bit)
  - Require recompilation (Stackguard, /GS)
  - Break backwards compatibility (ASLR)

- Need an approach to software security that is
  - Robust - no false positives on real-world code
  - Practical - works on unmodified binaries
  - Safe - few false negatives
  - Fast
DIFT: Dynamic Information Flow Tracking

- **DIFT taints** data from untrusted sources
  - Extra tag bit per word marks if untrusted

- **Propagate** taint during program execution
  - Operations with tainted data produce tainted results

- **Check** for suspicious uses of tainted data
  - Tainted code execution
  - Tainted pointer dereference (code & data)
  - Tainted SQL command

- **Potential:** protection from low-level & high-level threats
DIFT Example: Buffer Overflow

```
int buf[8];
for (i = 0; i < len; i++)
    buf[i] = u;
return;
```

Vulnerable C Code

![Diagram of tainted pointer dereference and security trap]

- Tainted pointer dereference $\Rightarrow$ security trap
Hardware DIFT Overview

- The basic idea [Suh’04, Crandall’04, Chen’05]
  - Extend HW state to include taint bits
  - Extend HW instructions to check & propagate taint bits

- Hardware advantages
  - Negligible runtime overhead
    - Software DIFT overheads range from 3-37x
  - Works with multithreaded and self-modifying binaries
  - Apply tag policies to OS
Raksha Overview & Features

Unmodified binaries

User 1
App Binary

User 2
App Binary

User 3
App Binary

Operating System
Tag Aware

Security Monitor

HW Architecture
Tags

- Cross-process info flow
- Save/restore tags
- Set security policies
- Control HW check/propagate
- 4 tag bits per word
- HW check/propagate
- User-level security traps
Outline

- Motivation & DIFT overview

- Preventing buffer overflows with DIFT
  - Previous work
  - Novel BOF prevention policy

- Evaluation
  - Prototype
  - Security experiments
  - Lessons learned

- Conclusions
Naïve Buffer Overflow Detection

- Previous DIFT approaches recognize **bounds checks**
  - Must bounds check untrusted info before dereference
    - Example: \( \text{if} \ (u < \text{len}) \ \text{print} \ \text{buf}[u]; \)

- **Taint** untrusted input
- **OR Propagate** taint on load, store, arithmetic, logical ops
- **Clear** taint on bounds checks
  - Comparisons against untainted info
- **Check** for tainted code, load/store/jump addresses
  - Forbid tainted pointer deref, code execution
Problems with Naïve Approach

- Not all bounds checks are comparisons
  - Example: `str++ = digits[val % 10]`
  - GCC, glibc, gzip…

- Not all comparisons are bounds checks
  - Example: `if (sz < fastbin_size) insert_fastbin(chunk);`
  - Resulted in false negative during traceroute/malloc exploit

- Bounds checks are not required for safety!
  - Example: `return isdigit[(unsigned char)x]`
    - isdigit array is 256 entries! Don’t need any bounds check
    - But stripped binary doesn’t tell us array sizes….

End result: unacceptable false positives in real code
Building a Better Security Model for BOF

- Buffer overflow attacks rely on injecting **pointers**
  - Code pointers
    - Return address, Global Offset Table (GOT), function ptr
  - Data pointers [Chen 05]
    - Filenames, permission/access control structures, etc

- Why pointers?
  - They’re everywhere!
    - Every stack frame (local pointers, frame pointer, ret addr)
    - Every free heap object (glibc)
    - Global Offset Table, constructors, destructors, …
  - Security-critical
    - Control pointers - arbitrary code execution
    - Data Pointers – subvert logic using tainted data structures
Preventing Pointer Injection with DIFT

- Buffer overflows exploits overwrite **pointers**
  - But should **never** receive pointer from network!
  - Tainted data used as pointer **index**, never as pointer **address**

- New DIFT BOF Policy
  - Tainted data cannot be dereferenced directly
  - Must be combined with application pointer to be safe
  - Pointer bit – tag legitimate application pointers
  - Taint bit – tag untrusted data

- But how do we identify legitimate application pointers?
New BOF Policy – Taint bit

- **Goal:** conservatively track untrusted information
  - Do **not** try to clear taint by recognizing bounds checks
  - Only clear taint when reg/mem word overwritten

- **Taint** untrusted input
- **OR Propagate** on load, store, arithmetic, logical ops
- **Check** for tainted code
- **Check** if code/data ptr is tainted and **not** a valid ptr
  - Security exception if Taint bit set & Pointer bit clear
New BOF Policy – Pointer bit

- **Propagate** Pointer bit during valid pointer ops
  - Load/Store Pointer
  - Pointer +,-,OR,AND Non-Pointer
  - Pointer +,- Pointer
    - Encountered in real-world, byte of pointer used as array index
- **Clear** P-bit on all other operations
  - Multiply, logical negation, etc
- **Check** for untrusted pointer dereferences
  - Security exception if T-bit set, P-bit clear
Identifying Userspace Pointers

- **Initialize** P-bit for all local variable references
  - Set P-Bit for stack pointer

- **Initialize** P-bit for all dynamically allocated memory references
  - Set P-bit for return value of mmap, brk syscalls

- **Initialize** P-Bit for static/global variable references
  - Scan all executable, library objects for these references
    - Scan both code, data regions
    - Set P-bit for potential any potential valid pointers
  - ABI (ELF, PE) restricts such references
    - Must be valid relocation entry type
BOF Protection in Kernel Space

- OS dereferences untrusted pointers!
  - System call arguments come from untrusted userspace
  - Example: `int unlink(const char * pathname)`

- Why is this safe?
  - All user pointers must be checked by `access_ok()`
  - Ensures user pointer is in userspace, not kernelspace

- What instructions may access userspace?
  - Any instruction accessing userspace may cause MMU fault
  - All modern Unix OSes build tables of these instructions!
    - Any MMU fault not found in the table is an OS bug

- Safe untrusted pointer dereference in Linux:
  - Tainted pointer must point to userspace
  - PC must be in MMU fault list
Raksha Prototype System

- Full-featured Linux system

- HW: modified Leon-3 processor
  - Open-source, Sparc V8 processor
  - Single-issue, in-order, 7-stage pipeline
  - Modified RTL for processor & system
  - Mapped to FPGA board (65Mhz workstation)

- SW: ported Gentoo Linux distribution
  - Based on 2.6 kernel (modified to be tag aware)
  - Kernel preloads security manager into each process
  - Over 14,000 packages in repository (GNU toolchain, apache, sendmail, …)
Experiments (Userspace)

- Successfully running Gentoo **without false positives**
  - Every program, even init, has BOF protection enabled
  - Run gcc, OpenSSH, sendmail, etc.
- Prevented attacks on real-world applications

<table>
<thead>
<tr>
<th>Program</th>
<th>Attack</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polymorph</td>
<td>Stack overflow</td>
<td>Tainted code ptr</td>
</tr>
<tr>
<td>Atphttpd</td>
<td>Stack overflow</td>
<td>Tainted code ptr</td>
</tr>
<tr>
<td>Sendmail</td>
<td>BSS overflow</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>Traceroute</td>
<td>Double free</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>Nullhttpd</td>
<td>Heap overflow</td>
<td>Tainted data ptr</td>
</tr>
</tbody>
</table>

*All userspace programs are unmodified binaries*
Experiments (Kernelspace)

- Protect entire Linux kernel from BOF
  - **First** comprehensive kernel buffer overflow protection
  - Even protect assembly code, device drivers, ctx switch
- Only observed one potential false positive
  - Caused by previously undiscovered security hole!
- Prevented real-world attacks on Linux kernel

<table>
<thead>
<tr>
<th>Subsystem</th>
<th>Vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>quota system call</td>
<td>User/Kernel pointer deref</td>
</tr>
<tr>
<td>i2o driver ioctl</td>
<td>User/Kernel pointer deref</td>
</tr>
<tr>
<td>moxa driver</td>
<td>BSS overflow</td>
</tr>
<tr>
<td>cm4040 driver</td>
<td>Heap overflow</td>
</tr>
<tr>
<td>sendmsg system call</td>
<td>Stack, Heap overflow</td>
</tr>
</tbody>
</table>
Comprehensive BOF protection

- Can some BOF vulnerabilities still be exploited?
  - Yes, if BOF doesn’t rely on pointer corruption
    - Authentication flag, user IDs, array/pointer offsets…
  - Rare, but possible – depends on application data structure layout, etc

- Combine multiple BOF protection policies for safety!
  - Attacker must evade all active policies to succeed
    - But must ensure all policies have no real-world false positives…
  - Policy #1: Bounds check BOF protection for control pointer only
    - Bounds check false positives only observed for data pointers
    - Prevents control pointer array offset overwrites
  - Policy #2: Red Zone bounds checking for heap
    - Tag begin/end of each heap object with Sandbox bit
    - Raise error if user attempts to load/store to sandbox’d address
    - Detects heap buffer overflows

- Use Raksha to run all policies concurrently (w/ Pointer BOF)
  - No false positives – tested in userspace and kernelspace
  - Verified new policies stop control pointer overwrites, heap overflows (resp.)
Conclusions

- Pointer-based BOF protection is practical
  - Prevents real-world buffer overflows – code/data pointer
    - No source code access, debugging info, etc required
  - No observed false positives
    - Tested GCC, Apache, OpenSSH, etc

- Protection can even be extended to OS
  - Full OS - FS, MM, device drivers, context switch, etc
  - Only potential false positive was a real security hole

- Compose multiple policies for best protection
  - Only miss an attack if it can evade all active policies
Questions?

- Want to use Raksha?
  - Go to [http://raksha.stanford.edu](http://raksha.stanford.edu)
  - Raksha port to Xilinx XUP board
    - $300 for academics
    - $1500 for industry
  - Full RTL + Linux distribution coming soon
Bonus round: Why not bounds checking?

- **Compatibility**
  - C was never meant to be bounds checked
    - Ex: optimized glibc() memchr() reads out of bounds
    - Context sensitive- Apache ap_alloc => malloc=>brk
  - Inline assembly, Multithreading
  - Dynamically loaded plugins, dynamically gen’d code
  - Closed-source libraries, objects in other languages

- **Cost – recompiling is expensive**
  - **Global** recompilation of all system libs is not happening
  - Just ask MS to recompile MFC…

- **Performance**
  - Overheads must be low (single digit) to drive adoption