Real-World Buffer Overflow Protection for 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
  - End-to-End
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: Memory Corruption

Vulnerable C Code

```c
char buf[1024];
strncpy(buf,input);//buffer overflow
```
DIFT Example: Memory Corruption

Vulnerable C Code

char buf[1024];
strcpy(buf,input);//buffer overflow

\[
\begin{align*}
\text{r1} & \leftarrow \text{r1} + 4 \\
\text{load } \text{r2} & \leftarrow M[\text{r1}] \\
\text{store } M[\text{r3}] & \leftarrow \text{r2} \\
\text{jmp } M[\text{retaddr}] & \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1:</td>
<td>input+1020</td>
</tr>
<tr>
<td>r2:</td>
<td>0</td>
</tr>
<tr>
<td>r3:</td>
<td>buf+1024</td>
</tr>
<tr>
<td>retaddr:</td>
<td>safe</td>
</tr>
</tbody>
</table>
DIFT Example: Memory Corruption

Vulnerable C Code

```c
char buf[1024];
strcpy(buf,input);//buffer overflow
```

```
r1 ← r1 + 4
load r2 ← M[r1]
store M[r3] ← r2
jmp M[retaddr]
```

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1: input+1024</td>
</tr>
<tr>
<td></td>
<td>r2: 0</td>
</tr>
<tr>
<td></td>
<td>r3: buf+1024</td>
</tr>
<tr>
<td></td>
<td>retaddr: safe</td>
</tr>
</tbody>
</table>
DIFT Example: Memory Corruption

Vulnerable C Code

```c
char buf[1024];
strcpy(buf,input); //buffer overflow
```

```
r1 ← r1 + 4
load r2 ← M[r1]
store M[r3] ← r2
jmp M[retaddr]
```

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>r1:</td>
<td>input+1024</td>
</tr>
<tr>
<td>r2:</td>
<td>bad</td>
</tr>
<tr>
<td>r3:</td>
<td>buf+1024</td>
</tr>
<tr>
<td></td>
<td>retaddr:</td>
</tr>
<tr>
<td></td>
<td>safe</td>
</tr>
</tbody>
</table>
DIFT Example: Memory Corruption

Vulnerable C Code

```c
char buf[1024];
strcpy(buf,input);//buffer overflow
```

```
r1 ← r1 + 4
load r2 ← M[r1]
store M[r3] ← r2
jmp M[retaddr]
```

<table>
<thead>
<tr>
<th>T</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r1: input+1024</td>
</tr>
<tr>
<td></td>
<td>r2: bad</td>
</tr>
<tr>
<td></td>
<td>r3: buf+1024</td>
</tr>
<tr>
<td></td>
<td>retaddr: bad</td>
</tr>
</tbody>
</table>
DIFT Example: Memory Corruption

Vulnerable C Code

```c
char buf[1024];
strcpy(buf,input);//buffer overflow
```

```
r1 ← r1 + 4
load r2 ← M[r1]
store M[r3] ← r2
jmp M[retaddr]
```

<table>
<thead>
<tr>
<th></th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>r1: input+1024</td>
</tr>
<tr>
<td></td>
<td>r2: bad</td>
</tr>
<tr>
<td></td>
<td>r3: buf+1024</td>
</tr>
<tr>
<td></td>
<td>retaddr: bad</td>
</tr>
</tbody>
</table>

- Tainted pointer dereference ⇒ security trap
Hardware DIFT Overview

- The basic idea [Suh’04, Crandall’04, Chen’05, Dalton ‘07]
  - Extend HW state to include taint bits
  - Extend HW instructions to check & propagate taint bits

- Hardware Advantages
  - Negligible runtime overhead
  - Works with multithreaded and self-modifying binaries
  - Apply tag policies to OS
Raksha Overview & Features [Dalton ’07]

Unmodified binaries

- User 1
  - App Binary
- User 2
  - App Binary
- SysAdmin
  - Security Manager

- Set security policies
  - Control HW check/propagate
  - Further SW analysis

- Operating System
  - Tag Aware

- Cross-process info flow
  - Save/restore tags

- HW Architecture
  - Tags

- 4 tag bits per word
- HW check/propagate
- User-level security traps
Check Policy Example: \texttt{load}

\texttt{load \ r2 \leftarrow M[r1+\text{offset}]}

\textbf{Check Enables}

1. Check source register
   \texttt{If Tag(r1)==1 \ then \ security\_trap}

2. Check source address
   \texttt{If Tag(M[r1+\text{offset}])==1 \ then \ security\_trap}

Both enables may be set simultaneously
**Propagate Policy Example: load**

\[
\text{load } r2 \leftarrow M[r1+offset]
\]

**Propagate Enables**

1. Propagate only from source register
   \[\text{Tag}(r2) \leftarrow \text{Tag}(r1)\]

2. Propagate only from source address
   \[\text{Tag}(r2) \leftarrow \text{Tag}(M[r1+offset])\]

3. Propagate only from both sources
   \[\begin{align*}
   &\text{OR mode: Tag}(r2) \leftarrow \text{Tag}(r1) \lor \text{Tag}(M[r1+offset]) \\
   &\text{AND mode: Tag}(r2) \leftarrow \text{Tag}(r1) \land \text{Tag}(M[r1+offset]) \\
   &\text{XOR mode: Tag}(r2) \leftarrow \text{Tag}(r1) \oplus \text{Tag}(M[r1+offset])
   \end{align*}\]
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

- 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, …)
Outline

- Motivation & DIFT overview

- Preventing Buffer Overflows with DIFT
  - Previous Work
  - Novel DIFT buffer overflow prevention policy

- Evaluation
  - Security experiments
  - Lessons learned

- Conclusions
Naïve Buffer Overflow Detection

- Previous DIFT approaches recognize **bounds checks**
  - Must bounds check untrusted information to dereference

- **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
  - `*str++ = digits[val % 10]` (glibc)
  - `ent = hashtbl[x & TABLESZ - 1]` (GCC)

- Not all comparisons are bounds checks
  - If `(chunksize(sz) < FASTBIN_SZ)
    - `malloc()` code caused false negative in traceroute exploit

- Bounds checks are not required for safety!
  - `return isdigit[(unsigned char)x]` (glibc)
    - `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
Preventing BOF with Pointer Identification

- New approach: prevent attackers from injecting pointers
  - Tainted information should not be directly dereferenced
  - Instead, use as offset combined with legitimate pointer

- Buffer overflow attacks rely on injecting pointers
  - Pointers are everywhere and security-critical
  - Code pointers (return address, function pointer, global offset table)
  - Data pointers (malloc heap chunks, filenames, permission structures)

- DIFT policy based on Pointer Injection
  - Track untrusted data (Taint bit) and legitimate pointers (Pointer bit)
    - Use two separate DIFT analyses
  - 2 tag bits per word – T bit, P-bit
  - Untrusted data may only be used an index to a legitimate pointer
    - Forbid any dereference with T-bit set and P-bit clear
New Policy for Taint Bit

- Goal: conservatively track untrusted information
  - Don’t try to clear taint by recognizing bounds checks
  - Only clear when reg/mem word overwritten by clean data

- **Taint** untrusted input

- **OR Propagate** on load, store, arithmetic, logical ops

- **Check** on code execution
  - Trap if code is tainted

- **Check** on load/store/jump address
  - Trap if address is tainted but does not have P-bit set
New Policy for Pointer Bit

- Goal: Identify all valid pointers at runtime

- **Initialize** P-bit for pointers to statically allocated mem at startup
  - More details on next slide on how to identify these

- **Initialize** P-bit for all pointers to dynamically allocated mem
  - Return value of mmap, shmat, brk syscalls

- **Propagate** P-bit during valid pointer ops
  - Load/Store Pointer
  - Pointer +,- Non Pointer
  - Pointer +,-, OR Pointer
    - Rare corner case in gcc, fprintf("%ld", pointer) …
  - Pointer AND non-pointer (only if pointer alignment)
  - Clear P-bit on all other operations
Identify Pointers at Startup

- Must set P-bit for all regs, memory with valid pointer at startup
  - Only regs with valid pointer are Stack Pointer, PC

- Scan Data and Code of all Objects (Executable and Libraries)
  - Set P-bit for potential valid pointers

- Object File Format (ELF, PE, etc) restricts references
  - Any reference to statically allocated mem must be relocatable
  - Only a few supported relocation entry formats...
  - Makes recognizing pointers in code/data practical
Identify Pointers cont’d

- Identifying Pointers in Data Segments
  - ELF, PE restrict data references to symbol + offset
    - Valid: `int * y = &x + 12`
    - Invalid: `int * y = &x >> 12`
  - Identify word of data as a pointer if
    - `ObjectFile_Start <= word < ObjectFile_End`

- Identifying Pointers in Code Segments
  - ELF SPARC restricts code references to sethi/or pairs
  - sethi instruction used to set upper 22 bits of register
  - Set P bit of sethi insn if constant within current obj file
  - At runtime, P-bit of sethi instructions propagates to dest
Protecting the Linux Operating System

- P-Bit, T-Bit initialization similar to userspace
  - OS has hardcoded pointer constants for heaps, I/O regions

- Problem: OS dereferences untrusted pointers!
  - System call arguments are untrusted
  - ssize_t write(int fd, const void * buf, size_t count)
  - Kernel must dereference buf, even though it is untrusted

- New security requirements
  - Must allow legitimate, safe user pointer dereferences
  - Must forbid user pointers into kernelspace
    - User/Kernel pointer dereference attack (compromises OS)
Protecting Linux cont’d

- **Solution: __ex_table**
  - Only user pointer dereferences cause MMU faults
  - __ex_table lists all instructions that may MMU fault
  - Similar data structures exist in Free/Net/OpenBSD, Solaris

- **Preventing kernel memory corruption**
  - Security exception if dereference tainted pointer
  - Exception handler permits tainted deref only if
    - PC is found in __ex_table
    - Load/store address is in userspace
  - Prevents buffer overflows and user/kernel pointer deref

- **Found one local DoS bug with this technique**
  - See paper for more details
Experiments

- Successfully running Gentoo on Raksha
  - Full FPGA-based prototype
  - Modern Linux distribution
  - Run gcc, OpenSSH, sendmail, Apache, etc.

- Protecting all of Userspace
  - Every program, every instruction
  - Policy enforced by trusted userspace monitor

- Protecting Kernel Space
  - Everything but first few instructions of trap handler
    - These instructions enable BOF tag policy
  - Protect bootup code, optimized handwritten assembly, context switching code, etc
# Userspace Buffer Overflow Results

<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>Atphtpd</td>
<td>Stack overflow</td>
<td>Tainted code ptr</td>
</tr>
<tr>
<td>Nullhtpd</td>
<td>Heap overflow</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>Traceroute</td>
<td>Double free</td>
<td>Tainted data ptr</td>
</tr>
<tr>
<td>Sendmail</td>
<td>BSS overflow</td>
<td>Tainted data ptr</td>
</tr>
</tbody>
</table>

All applications are unmodified binaries
No false positives
## Kernelspace Buffer Overflow Results

<table>
<thead>
<tr>
<th>Module</th>
<th>Attack</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quotactl syscall</td>
<td>User/Kernel Pointer</td>
<td>User pointer to OS data</td>
</tr>
<tr>
<td>l2o driver</td>
<td>User/Kernel Pointer</td>
<td>User pointer to OS data</td>
</tr>
<tr>
<td>Sendmsg syscall</td>
<td>Stack, Heap Overflow</td>
<td>Tainted data pointer</td>
</tr>
<tr>
<td>Moxa driver</td>
<td>BSS Overflow</td>
<td>Tainted data pointer</td>
</tr>
<tr>
<td>Cm4040 driver</td>
<td>Heap Overflow</td>
<td>Tainted data pointer</td>
</tr>
</tbody>
</table>

Protection enabled for all of kernelspace

No false positives
Conclusions

- Bounds check recognition is fatally flawed
  - Diversity of operations is immense (e.g. % on SPARC)
  - Don’t even need to bounds check in some corner cases
    - Cannot disambiguate these cases from attacks in practice

- New BOF policy – prevent pointer injection
  - Track tainted data and legitimate application pointers
  - Forbid dereference if T bit set and P-bit clear

- Result: protect code and data pointer with no false positives
  - Prevented attacks in userspace, kernelspace
  - Verified no false positives in user/kernel
    - Ran Apache, GCC, mysql, etc
    - Untrusted sources should never supply pointers
Further Information in the Paper

- Prototype implementation description
  - Full summary of check, propagate modes, etc

- Portability discussions
  - How to port T-bit, P-bit rules to x86
  - How to apply Linux kernel BOF rules to BSDs, Solaris

- Additional DIFT policies
  - Provide better coverage by using multiple policies
  - Red Zone Bounds Checking
  - Bounds Check Recognition for control pointers only
  - Format string protection
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