Diversified Process Replicae for Defeating Memory Error Exploits

Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi
<sullivan@security.dico.unimi.it>
<lorenzo@cs.sunysb.edu>

Università degli Studi di Milano, Italy
visiting scholar at SUNY at Stony Brook, NY, USA

3rd Workshop on Information Assurance

13th April 2007
Table of Contents

1. Memory Error
2. Artificial Diversity
3. Diversified Process Replicaæ
4. Effectiveness
5. Practical Issues
   - Shared Memory Management
   - Signals & Threads
6. Experimental Results
7. Future Works
Memory Error

The Issue

A memory error occurs when an object accessed using a pointer expression is different from the one intended.

- out-of-bounds access (e.g., buffer overflow)
- access using a corrupted pointers (e.g., buffer overflow, format bug)
- uninitialized pointer access;
- dangling pointers;
- . . .
The Exploit

Memory Error Exploits

- well-known way to subvert/divert a legal process execution flow
- usually overwrite control-data with *absolute known* values:
  - saved return addresses
  - application-specific function pointers
  - “other” function pointers (e.g., GOT, .dtors, C++ vrt ptrs)
e.g., stack/data buffer overflow, format string bug, malloc chunk exploit, integer overflow
Artificial Diversity
State of the Art

Biological Diversity
Plays a crucial role for the survivability of every biological species

- a successful memory error exploit usually relies on using well-known absolute memory addresses

⇒ solution: make systems appear different!

- Address Space Layout Randomization
- Address Space Obfuscation
- Instruction Set Randomization
Artificial Diversity
Limitations of the State of the Art

Usually diversity is applied on a process itself, but it:

- requires high entropy
- relies on keeping secrets:
  - disclosed by information leakage attacks
  - defeated by brute forcing attacks
- generally cannot defeat partial memory overwriting attacks (e.g., Impossible Path Executions)
- cannot defeat memory error exploits with certainty
- so far, offers a probabilistic protection mechanism
Our Approach: Diversified Process Replicæ

Framework

- \( T \), the replicator & monitoring process, creates \( P_r \), a replica of the protected process \( P \)
- \( T \) makes \( P \) and \( P_r \) to behave identically on benign input
- \( P \) and \( P_r \) are properly diversified to detect behavioral divergence caused by malicious input, i.e., memory error attacks
Our Approach: Diversified Process Replicæ

Process Replication

\( T \) synchronizes \( P \) and \( P_r \) for every system call invocation (\textit{rendez-vous} point), and:

- checks for system call consistency (e.g., system call arguments, system call number)
- \textit{simulates} certain system calls (e.g., \texttt{read}, \texttt{write}, \texttt{recv}, \texttt{send})
  - replicates input, correctly handles output on I/O system calls
  - performs system call “once”
  - returns consistent results to \( P \) and \( P_r \)
- lets \( P \) and \( P_r \) \textit{execute} others system calls (e.g., \texttt{brk}, \texttt{signal})
- carefully treats other system calls (e.g., \texttt{mmap2}, \texttt{shmat}, \texttt{shmget})
Our Approach: Diversified Process Replicæ

Process Diversification

- non-overlapping address spaces to combat memory corruption attacks targeting absolute memory address
- address space shifting to combat partial overwriting memory corruption attacks

⇒ both address non relative control-data memory error exploits and some non-control data

- statically: custom linker script which takes care of the executable .text, .data, .bss, heap (next to .bss)
- dynamically: with a modified ld-linux.so which takes care of the executable stack and shared objects “relocation”
Our Approach: Diversified Process Replicæ
Address Space Partitioning

D. Bruschi, L. Cavallaro, and A. Lanzi
Effectiveness

Stack-based Buffer Overflow

(a) (b)

0xbfff1234
higher addresses (stack growth)

0xbfff1234
lower addresses

Process Stack

arguments

0xbfff1234

SFP (overwrit)

Injected Malicious Code

Process Replica Stack

Unmapped Area (segfault)

0xbfff1234
arguments

0xbfff1234
SFP (overwrit)

Injected Malicious Code

0x7fff1245
Unmapped Area (segfault)
Practical Issues

- shared memory management
- signals
- threads
Shared Memory
mmap-based and “classical” shared memory

mmap-based

1. non-anonymous
   (a) private mapping (intra-process communication)
   (b) shared mapping (*inter-process communication*)

2. anonymous (intra-process communication)

classical shared memory

(a) private mapping (intra-process communication)
(b) shared mapping (*inter-process communication*)
**Shared Memory**

Data inconsistency and Behavioral Divergence

- $P$ and $P_r$ create a readable and writable non-anonymous shared memory segment $M$
- $ptr[0]$ points to the beginning of $M$

```plaintext
1 if (ptr[0] == 'A')
  2   ptr[0] = 'B';
3 else
  4     ptr[0] = 'C';
5 ...
6 /*
7   * process invokes system calls based on the
8   * value held by ptr[0]
9   */
```
let suppose that only $P$ and $P_r$ are sharing a resource $R$

as seen before, $P$ and $P_r$ start an unwanted form of *inter-process communication* between them

the direct consequence is that $P$ and $P_r$ might exhibit a different behavior and $R$ might be inconsistent

the solution is simple: let $P_r$ create a *private* mapping, i.e., no IPC between $P$ and $P_r$
**Assumption**

“[...] What is normally required [when using shared memory], however, is some form of synchronization between the processes that are storing and fetching information to and from the shared memory region”

- the scenario with unrelated processes is more tricky
- creating a *private* mapping is *necessary* but it is *not sufficient*
- an external process \( E \) might modify the resource
  - \( P \) will see the modification (shared mapping)
  - \( P_r \) will not (private mapping)
- \( P_r \) must operate on an *up-to-dated* view of the shared resource \( R \)
the Assumption provides the following:

- it makes possible to decide \textit{when} to perform the refresh operation (rendez-vous point)
- it permits to wait for \( P \) to “acquire a lock” for \( R \): it grants data consistency during the \textit{refresh} operation

Main point: \textit{how} and \textit{when} to update the memory regions where \( R \) is referenced at:

- to get “\textit{when}” requires to analyze the synchronization mechanisms \( P \) can use
- knowing such mechanisms help to find the answer to the “\textit{how}” \( \Rightarrow \) \textit{Fault Interpretation}
Fault Interpretation

- $T$ marks $P$ and $P_r$ shared mapping as read-only
- $T$ exploits the CPU page fault exception to know whenever $P$ is writing into a shared memory area
- $T$ interprets the outcome of the synchronization adopted (might be tricky)
- $T$ refreshes $P_r$ shared memory mapping if $P$ acquired the lock successfully
signals are asynchronous events; they might cause $P$ and $P_r$ to behave differently if delivered asynchronously to them
  - signals can be delivered synchronously by postponing them at the next rendez-vous point (in general)

threads share the same issues raised by shared memory management, but their treatment could be more tricky
  - open issue if shared control-dependencies data might modify a thread’s behavior
  - scheduling $P$ and $P_r$ threads in the same way might not be enough
user space ptrace prototype on a Debian GNU/Linux system, 2.6.x kernel

clone/fork/vfork support (still unreliable)

shared memory management (preliminary idea)

signals management (preliminary idea)

preliminary experimental results (100 conns, 4 sess x conn, 13 reqs x conn, ~ 7.5MB web site):

<table>
<thead>
<tr>
<th>#</th>
<th>Throughput</th>
<th>MB/s (real)</th>
<th>MB/s (DPR)</th>
<th>slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>thttpd (mmap)</td>
<td>12386.9</td>
<td>12238.8</td>
<td>1.20%</td>
</tr>
<tr>
<td>2</td>
<td>thttpd (mmap-nocache)</td>
<td>12718.4</td>
<td>12496.5</td>
<td>1.75%</td>
</tr>
<tr>
<td>3</td>
<td>thttpd (read)</td>
<td>12599.5</td>
<td>12117.4</td>
<td>3.83%</td>
</tr>
<tr>
<td>4</td>
<td>thttpd (read-nocache)</td>
<td>12603.7</td>
<td>7086.3</td>
<td>43.78%</td>
</tr>
<tr>
<td>5</td>
<td>thttpd (read-nocache-single)</td>
<td>9134.5</td>
<td>2838.1</td>
<td>68.93%</td>
</tr>
</tbody>
</table>
Future Works

- full implementation of the prototype
- assess the viability and practicability of the shared memory solution
- improve protection from partial overwriting memory corruption attacks targeting control-data
- address relative addressing and, in general, non-control-data memory corruption attacks
- performance:
  - hybrid system call interposition implementation
  - (selective) file system replication (currently testing)
  - could SMP help out?

It seems to be an exciting research topic! :-(
Questions & Answers

Q & A?
Thank you! :-)

D. Bruschi, L. Cavallaro, and A. Lanzi

Diversified Process Replicæ