Diversified Process Replicæ 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 Replicæ
4. Effectiveness
5. Practical Issues
    - Shared Memory Management
    - Signals & Threads
6. Experimental Results
7. Future Works

Memory Error

The Issue

Memory Error

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;
- ...

Memory Error

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

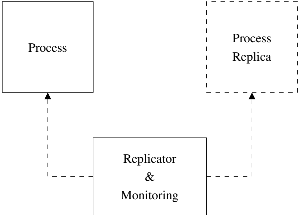
Limitations of the State of the Art

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 (*rendez-vouz* point), and:

- checks for system call consistency (e.g., system call arguments, system call number)
- *simulates* certain system calls (e.g., `read`, `write`, `recv`, `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 *execute* others system calls (e.g., `brk`, `signal`)
- carefully treats other system calls (e.g., `mmap2`, `shmat`, `shmget`)

Our Approach: Diversified Process Replicæ

Process Diversification

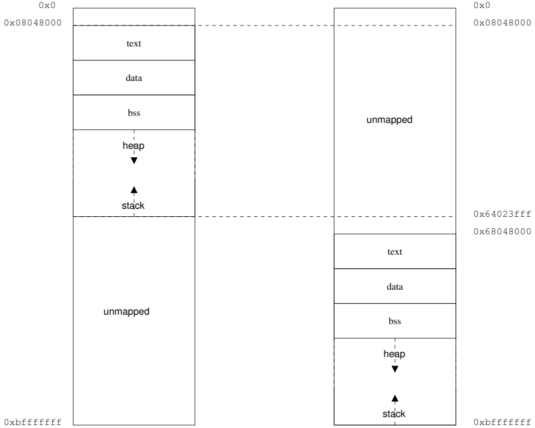
- non-overlapping address spaces to combat memory corruption attacks targeting absolute memory address
- address space shifting to combat partial overwriting memory corruption attacks

⇒ 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"

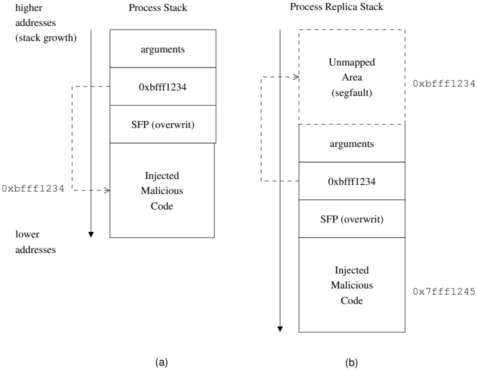
Our Approach: Diversified Process Replicæ

Address Space Partitioning



Effectiveness

Stack-based Buffer Overflow



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 $\mathcal{M}$
- `ptr[0]` points to the beginning of $\mathcal{M}$

```
if (ptr[0] == 'A')
   ptr[0] = 'B';
else
   ptr[0] = 'C';
...
/*
 * process invokes system calls based on the
 * value held by ptr[0]
*/
```

Shared Memory

Related-only Processes

- 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

Shared Memory

Unrelated Processes (1)

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

Shared Memory

Unrelated Processes (2)

- the Assumption provides the following
  - it makes possible to decide *when* to perform the refresh operation (rendez-vouz point)
  - it permits to wait for P to "acquire a lock" for R: it grants data consistency during the *refresh* operation

Main point: *how* and *when* to update the memory regions where R is referenced at:

- to get "*when*" requires to analyze the synchronization mechanisms P can use
- knowing such mechanisms help to find the answer to the "*how*" ⇒ *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 and Non-Determinism

- 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-vouz 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

Diversified Process Replicæ

Project Status & Experimental Results

- 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, $\sim$ 7.5MB web site):

#	Throughput	MB/s (real)	MB/s (DPR)	slowdown
1	`thttpd` (mmap)	12386.9	12238.8	1.20%
2	`thttpd` (mmap-nocache)	12718.4	12496.5	1.75%
3	`thttpd` (read)	12599.5	12117.4	3.83%
4	`thttpd` (read-nocache)	12603.7	7086.3	43.78%
5	`thttpd` (read-nocache-single)	9134.5	2838.1	68.93%

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! :-)