Comprehensive Memory Error Protection via Diversity and Taint-Tracking

Dipartimento di Informatica e Comunicazione
Università degli Studi di Milano, Italy

February, 14 2008

PhD Dissertation Defense

Adviser: Prof. R. Sekar
Co-Adviser: Prof. D. Bruschi

PhD Candidate
Lorenzo Cavallaro

Table of Contents

Motivation

Memory Error
- Research Goal

State of the Art
- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches
- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

Related Works

Future Directions

Conclusions

Table of Contents

Motivation

Memory Error
- Research Goal

State of the Art
- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches
- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

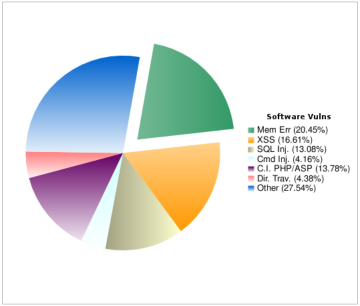
Related Works

Future Directions

Conclusions

Motivation

Breakdown of the NVD NIST Software Security Vulnerabilities (2006 – Q1-3 2007)



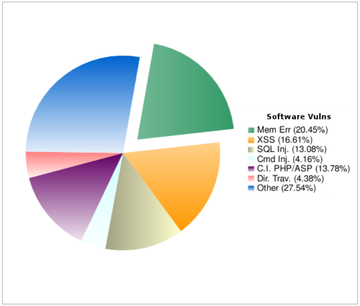
- Memory errors are still a relevant issue
- Most effective countermeasures are
  - Attack-specific
  - Mainly probabilistic
  - Vulnerable to alternative attacks

Our result:

- *Comprehensive* solutions
- Mainly *deterministic* protection
- *Resilient* to most evasions

Motivation

Breakdown of the NVD NIST Software Security Vulnerabilities (2006 – Q1-3 2007)



- Memory errors are still a relevant issue
- Most effective countermeasures are
  - Attack-specific
  - Mainly probabilistic
  - Vulnerable to alternative attacks

Our result:

- *Comprehensive* solutions
- Mainly *deterministic* protection
- *Resilient* to most evasions

Table of Contents

Motivation

Memory Error
- Research Goal

State of the Art
- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches
- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

Related Works

Future Directions

Conclusions

Memory Error

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

- 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 exploitation generally relies on

- Data corruption
- Gathering information on memory location addresses

Memory Error

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

- 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 exploitation generally relies on

- Data corruption
- Gathering information on memory location addresses

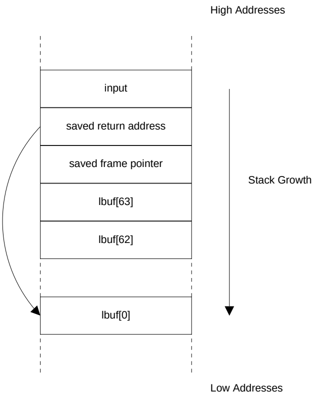
Memory Error I
Examples

Code pointer corruption

```
int foo(char *input) {
  char lbuf[64];
  int i;

  for (i = 0; i < strlen(input); i++)
    lbuf[i] = input[i];

  return 0;
```



Memory Error II

Examples

Data pointer corruption

```
FILE * getdatasock(char *arg1, ...) {
    char buf[128];
    ...
    seteuid(0);
    setsockopt(...);
    sprintf(buf, arg1);
    ...
    seteuid(pw->pw_uid);
}
```

Data corruption

```
void write_user_data(void) {
    FILE * fp ;
    char user_filename[256], user_data[256];

    gets(user_filename);

    if (privileged_file(user_filename)) {
      fprintf(stderr, "Illegal filename. Exiting.\n");
      exit(1);
    } else {
      gets(user_data); // overflow into user_filename
      fp = fopen(user_filename, "w");
      if (fp) {
        fprintf(fp, "%s", user_data);
        fclose(fp);
      }
    }
}
```

Research Goal

Program transformation techniques for memory error protection

- Comprehensive
- Mainly deterministic
- Vulnerability and attack-independent
- Resilient to different evasions

Table of Contents

Motivation

Memory Error
- Research Goal

State of the Art
- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches
- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

Related Works

Future Directions

Conclusions

Artificial Diversity

Biological Diversity

Plays a crucial role for the survivability of every biological species

- Memory error exploits rely on using *well-known* memory addresses

$\Rightarrow$ Make systems appear different!

- Address Space Layout Randomization (ASLR) [15]
- Fine-grained Address Space Randomization (ASR) [12, 11]
- Instruction Set Randomization (ISR) [3]

Artificial Diversity

Biological Diversity

Plays a crucial role for the survivability of every biological species

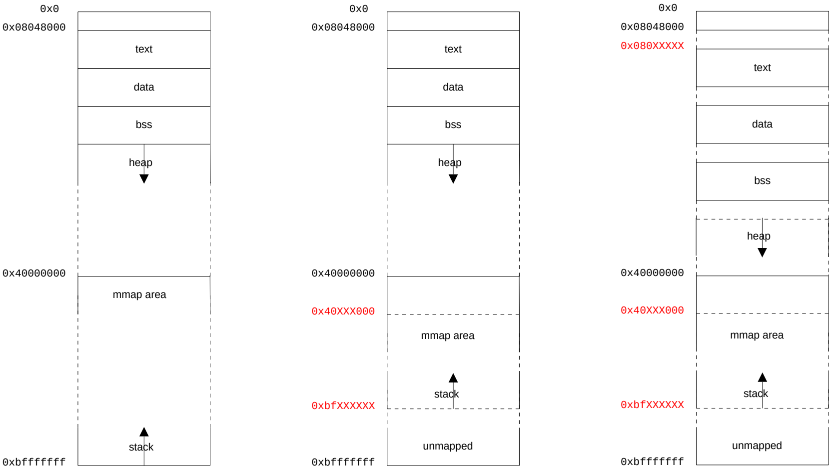
- Memory error exploits rely on using *well-known* memory addresses

$\Rightarrow$ Make systems appear different!

- Address Space Layout Randomization (ASLR) [15]
- Fine-grained Address Space Randomization (ASR) [12, 11]
- Instruction Set Randomization (ISR) [3]

Artificial Diversity

Examples: ASLR [15] & Fine-grained ASR [12]



Artificial Diversity
Limitations

Diversity applied on a process itself

- Requires high entropy
- Relies on keeping secrets
  - ... Disclosed by information leakage attacks [13]
  - ... Defeated by brute forcing attacks [6]
- Hard to counteract
  - Partial memory overwriting attacks
  - Most arbitrary data corruption
- Provides *probabilistic* protection

Artificial Diversity

Limitations

Diversity applied on a process itself

- Requires high entropy
- Relies on keeping secrets
  - ... Disclosed by information leakage attacks [13]
  - ... Defeated by brute forcing attacks [6]
- Hard to counteract
  - Partial memory overwriting attacks
  - Most arbitrary data corruption
- Provides *probabilistic* protection

Artificial Diversity
Limitations

Diversity applied on a process itself

- Requires high entropy
- Relies on keeping secrets
  - . . . Disclosed by information leakage attacks [13]
  - . . . Defeated by brute forcing attacks [6]
- Hard to counteract
  - Partial memory overwriting attacks
  - Most arbitrary data corruption
- Provides *probabilistic* protection

Artificial Diversity
Limitations

Diversity applied on a process itself

- Requires high entropy
- Relies on keeping secrets
  - ... Disclosed by information leakage attacks [13]
  - ... Defeated by brute forcing attacks [6]
- Hard to counteract
  - Partial memory overwriting attacks
  - Most arbitrary data corruption
- Provides *probabilistic* protection

Taint Analysis

Determines whether the value of a variable x is influenced by the value of another variable y

- It tracks how a program *untrusted* data (input) *flow* into *sinks* (output), security sensitive points
  - $x := y$ (explicit data-dependent flow)
  - **if** $x = k$ **then** $y = k^{'}$ (explicit control-dependent flow)

↑ It enforces taint-enhanced security policies on sinks to detect improper usage of *tainted* data

- *Code pointer* memory error corruption

↓ Hard or impossible to manually specify policy for some (memory error) vulnerabilities (FPs/FNs)

Taint Analysis

Determines whether the value of a variable x is influenced by the value of another variable y

- It tracks how a program *untrusted* data (input) *flow* into *sinks* (output), security sensitive points
  - $x := y$ (explicit data-dependent flow)
  - **if** $x = k$ **then** $y = k^{'}$ (explicit control-dependent flow)

↑ It enforces taint-enhanced security policies on sinks to detect improper usage of *tainted* data

- *Code pointer* memory error corruption

↓ Hard or impossible to manually specify policy for some (memory error) vulnerabilities (FPs/FNs)

Taint Analysis

Determines whether the value of a variable x is influenced by the value of another variable y

- It tracks how a program *untrusted* data (input) *flow* into *sinks* (output), security sensitive points
  - $x := y$ (explicit data-dependent flow)
  - **if** $x = k$ **then** $y = k'$ (explicit control-dependent flow)

↑ It enforces taint-enhanced security policies on sinks to detect improper usage of *tainted* data

- *Code pointer* memory error corruption

↓ Hard or impossible to manually specify policy for some (memory error) vulnerabilities (FPs/FNs)

Anomaly Detection

Determines whether a process behavioral profile $\mathcal{M}'$ is *consistent* with the behavioral profile $\mathcal{M}$ learnt during a *learning* or *training* phase

- Deviation from $\mathcal{M}$ observed during a *detection* phase are considered anomalous
- Anomalous events are considered as attacks' manifestations

↑ It *automatically* infers policies of legitimate process behaviors

- It detects *unknown* attacks

↓ High false positives (FPs) rate

- Training *not exhaustive* flags some unseen — but legitimate — behaviors as anomalous

Anomaly Detection

Determines whether a process behavioral profile $\mathcal{M}'$ is *consistent* with the behavioral profile $\mathcal{M}$ learnt during a *learning* or *training* phase

- Deviation from $\mathcal{M}$ observed during a *detection* phase are considered anomalous
- Anomalous events are considered as attacks' manifestations

↑ It *automatically* infers policies of legitimate process behaviors
  - It detects *unknown* attacks

↓ High false positives (FPs) rate
  - Training *not exhaustive* flags some unseen — but legitimate — behaviors as anomalous

Anomaly Detection

Determines whether a process behavioral profile $\mathcal{M}'$ is *consistent* with the behavioral profile $\mathcal{M}$ learnt during a *learning* or *training* phase

- Deviation from $\mathcal{M}$ observed during a *detection* phase are considered anomalous
- Anomalous events are considered as attacks' manifestations

↑ It *automatically* infers policies of legitimate process behaviors
  - It detects *unknown* attacks

↓ High false positives (FPs) rate
  - Training *not exhaustive* flags some unseen — but legitimate — behaviors as anomalous

Table of Contents

Motivation

Memory Error
- Research Goal

State of the Art
- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches
- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

Related Works

Future Directions

Conclusions

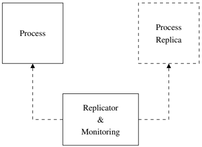
Diversified Process Replicæ

Framework

Idea

To couple the concept of *artificial diversity* and *process replication*

- T, the tracer, creates P_r, a *replica* of P
- T makes P and P_r to behave identically on benign input
- P and P_r are *artificially diversified*
  - $\Rightarrow$ Detect behavioral divergence caused by malicious input (i.e., memory error attacks)



Process Replication

Rendez-vouz

T synchronizes P and P_r at every system call invocation

- T checks for system call consistency (e.g., system call arguments, system call number)
- T *simulates* certain system calls (e.g., `read`, `send`)
  - It replicates input and handles output on I/O system calls
  - It performs the system call *once*
  - It returns consistent results to P and P_r
- T let P and P_r to *execute* other system calls (e.g., `brk`)
- T carefully handles other system calls (e.g., `mmap2`)

Process Replication

Rendez-vouz

T synchronizes P and P_r at every system call invocation

- T checks for system call consistency (e.g., system call arguments, system call number)
- T *simulates* certain system calls (e.g., `read`, `send`)
  - It replicates input and handles output on I/O system calls
  - It performs the system call *once*
  - It returns consistent results to P and P_r
- T let P and P_r to *execute* other system calls (e.g., `brk`)
- T carefully handles other system calls (e.g., `mmap2`)

Process Replication

Rendez-vouz

T synchronizes P and P_r at every system call invocation

- T checks for system call consistency (e.g., system call arguments, system call number)
- T *simulates* certain system calls (e.g., `read`, `send`)
  - It replicates input and handles output on I/O system calls
  - It performs the system call *once*
  - It returns consistent results to P and P_r
- T let P and P_r to *execute* other system calls (e.g., `brk`)
- T carefully handles other system calls (e.g., `mmap2`)

Process Diversification

- *Non-overlapping* address spaces for absolute overwriting
- Address space *shifting* for partial overwriting

Result

Code and data pointer corruption are defeated

Statically: custom linker script for `.text`, `.data`, `.bss`, base of heap

Dynamically: modified `ld-linux.so` for the executable stack and shared objects mapping

Process Diversification

- *Non-overlapping* address spaces for absolute overwriting
- Address space *shifting* for partial overwriting

Result

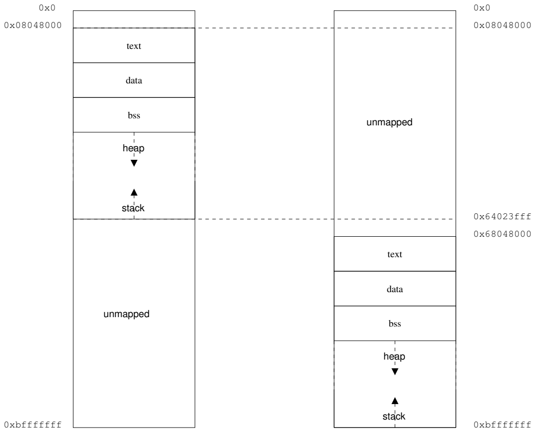
Code and data pointer corruption are defeated

Statically: custom linker script for `.text`, `.data`, `.bss`, base of heap

Dynamically: modified `ld-linux.so` for the executable stack and shared objects mapping

Process Replication

Address Space Partitioning



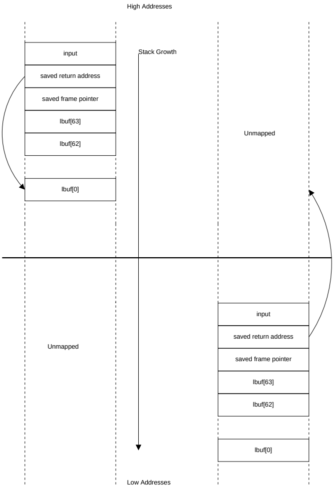
Effectiveness I

Code pointer corruption

```
int foo(char *input) {
  char lbuf[64];
  int i;

  for (i = 0; i < strlen(input); i++)
    lbuf[i] = input[i];

  return 0;
}
```



Effectiveness II

Limitations

It cannot thwart

- Arbitrary (non-pointer) data corruption
- Some information leakage

```
void write_user_data(void) {
    FILE *fp;
    char user_filename[256];
    char user_data[256];

    gets(user_filename);

    if (privileged_file(user_filename))
       exit(1);

    // overflow: corrupts user_filename

    gets(user_data);
    fp = fopen(user_filename, "w");
    if (fp) {
       fprintf(fp, "%s", user_data);
       fclose(fp);
    }
}
```

Experimental Results I
The Prototype

- User-space prototype developed on a Debian GNU/Linux system, 2.6.17 kernel, $5,700+$ LoC
- Modified run-time dynamic linker `ld-linux.so`
- Replication via `ptrace` implementation
- It supports
  - `clone`/`fork`/`vfork` support
  - Shared memory management
  - Signals management

Experimental Results
Throughput Penalties

100 conns, 4 sess/conn, 13 reqs/conn, $\sim$ 7.5MB web site

#	**Throughput**	**MB/s** (no DPR)	**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	$\sim$ 3.8%
4	`thttpd` (read-nocache)	12603.7	7086.3	$\sim$ 43.8%
5	`thttpd` (read-nocache-single)	9134.5	2838.1	$\sim$ 69%

Experimental Results
Latency Penalties

100 conns, 4 sess/conn, 13 reqs/conn, $\sim$ 7.5MB web site

#	**Latency**	**ms** (real system)	**ms** (DPR)	**slowdown**
1	`thttpd` (mmap)	3.5	4.6	31%
2	`thttpd` (mmap-nocache)	3.5	4.5	29%
3	`thttpd` (read)	3.5	5.3	51%
4	`thttpd` (read-nocache)	3.7	21.6	$\sim 6x$
5	`thttpd` (read-nocache-single)	166	646	$\sim 4x$

Table of Contents

Motivation

Memory Error

- Research Goal

State of the Art

- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches

- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

Related Works

Future Directions

Conclusions

Taint-enhanced Anomaly Detection

Idea

To couple *taint information* with learning-based *anomaly detection*

- Fine-grained taint analysis provides information about the ability of the attacker to *exercise* the vulnerability
- ↓ Hard to specify arbitrary security policies (FPs/FNs)
- Anomaly detection automatically learns application behaviors
- ↓ Learning approaches are *not exhaustive* (FPs/FNs)

$\Rightarrow$ Consider *tainted events* only

↑ False positives are decreased
↑ True positives are increased

Taint-enhanced Anomaly Detection

Idea

To couple *taint information* with learning-based *anomaly detection*

- Fine-grained taint analysis provides information about the ability of the attacker to *exercise* the vulnerability

↓ Hard to specify arbitrary security policies (FPs/FNs)

- Anomaly detection automatically learns application behaviors

↓ Learning approaches are *not exhaustive* (FPs/FNs)

⇒ Consider *tainted events* only

↑ False positives are decreased
↑ True positives are increased

Fine-grained Taint Analysis

Source-to-source *program transformation* technique

- It marks incoming input as untrusted (i.e., *tainted*)
- It tracks data propagation
- It inserts callback functions for every *sink* (e.g., system call)
  - Learning phase
  - Detection phase
  - Null-behavior (taint-enhanced only)

Learning-based Approaches

$$\Sigma = \{sinks\}, s(a_1, \cdots, a_n) \in \Sigma, a_i \; sinks \; arguments, i \in \{1, \cdots, n\}$$

- Context-sensitive analysis
- Taint information (e.g., a_i taintedness), $\forall s \in \Sigma$
- An event $s \in \Sigma$ is tainted if it exists at least one tainted a_i

For tainted events

Untainted bytes

- Longest common prefix (LCP)
- Minimum length

Tainted bytes

- Structural inference
- Maximum length

Learning-based Approaches

$$\Sigma = \{sinks\}, s(a_1, \cdots, a_n) \in \Sigma, a_i \ sinks \ arguments, i \in \{1, \cdots, n\}$$

- Context-sensitive analysis
- Taint information (e.g., a_i taintedness), $\forall s \in \Sigma$
- An event $s \in \Sigma$ is tainted if it exists at least one tainted a_i

For tainted events

Untainted bytes

- Longest common prefix (LCP)
- Minimum length

Tainted bytes

- Structural inference
- Maximum length

Effectiveness I

- Coarse-grained taint information
- Maximum length
- Structural inference

```
void write_user_data(void) {
    FILE *fp;
    char user_filename[256];
    char user_data[256];

    gets(user_filename);
    if (privileged_file(user_filename)) { exit(1); }
    gets(user_data);
    fp = fopen(user_filename, "w");
    if (fp) { fprintf(fp, "%s", user_data); fclose(fp); }
}
```

learning:

`user_data` taintedness & length

detection:

`user_data` length is violated during attack

Effectiveness II

Fine-grained taint information

```
FILE * getdatasock(char *arg1, ...) {
    char buf[128];
    ...
    seteuid(0);
    setsockopt(...);
// fmt bug overwrites current user cred
    sprintf(buf, arg1);
    ...
    seteuid(pw->pw_uid);
}
```

learning:

untainted `seteuid` argument

detection:

taintedness violation for `seteuid` argument `pw->pw_uid` during attack

Experimental Results
The Prototype

- Fine-grained taint analysis
  - CIL & OCaml for the program transformation ($\sim 5,000$ LoC)
  - C for the taint propagation strategy and callback insertion
- Learning-based anomaly detection approach
  - C/C++ for learning, detection and original behavior phase ($15,000+$ LoC)
  - Python for automatic code generation

Experimental Results I

#	App	# Traces (Learning)	# Traces (Detection)	FP	Overall FPs
1	`proftpd`	$68,851$	$983,740$	200	2.0×10^{-4}
2	`apache`	$58,868$	$688,100$	2000	2.9×10^{-3}

Table: Overall False Positives.

#	App	Taint	LCP	Min	Struct Inf.	T. Max	Overall FPs
1	`proftpd`	3.0×10^{-5}	3.0×10^{-5}	0	1.4×10^{-4}	0	2.0×10^{-4}
2	`apache`	0	4.3×10^{-4}	0	2.4×10^{-3}	0	2.9×10^{-3}

Table: False Positives Breakdown.

Experimental Results II

#	App	Unkn. untaint. traces	Taint. of sinks args	FPs (taint inf.)
1	`proftpd`	2.1×10^{-4}	3.0×10^{-5}	2.4×10^{-4}
2	`apache`	4.3×10^{-4}	0	4.3×10^{-4}

Table: Unknown/Untainted Traces.

#	App	slowdown (taint)	slowdown (taint-learn)	slowdown (taint-detect)
1	`proftpd`	3.1%	5.9%	9.3%
2	`apache`	5.7%	10.3%	18.5%

Table: Throughput Slowdown.

Table of Contents

Motivation

Memory Error

- Research Goal

State of the Art

- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches

- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

Related Works

Future Directions

Conclusions

Related Works

- Artificial diversity [11, 12, 15]
- Taint analysis [2, 7, 10, 16]
- Learning-based anomaly detection techniques
  - system call sequences [5]
  - FSA [14]
  - call stack information [4]
  - statistical multi-model (e.g., bytes frequency, token presence, structural inference) [8, 9]
  - data-flow relationship [1]

Table of Contents

Motivation

Memory Error

- Research Goal

State of the Art

- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches

- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

Related Works

Future Directions

Conclusions

Diversified Process Replicæ

Optimizations
- Do not simulate FS-related system calls that operate on a FS-objects $\mathcal{O}$ unless $\mathcal{O}$ is shared
- SMP

Dynamic Binary Translation (QEMU)
- ↑ Does not require program recompilation
- Faster then a `ptrace` implementation
- Partial overwrite protection is lost

Program Transformation
- Insert non-overlapping gaps between buffer-like variables of P and P_r to thwart some data corruption (probabilistic protection)

Diversified Process Replicæ

Optimizations
- Do not simulate FS-related system calls that operate on a FS-objects $\mathcal{O}$ unless $\mathcal{O}$ is shared
- SMP

Dynamic Binary Translation (QEMU)
↑ Does not require program recompilation
Faster then a `ptrace` implementation
Partial overwrite protection is lost

Program Transformation ↑ ↓
- Insert non-overlapping gaps between buffer-like variables of P and P_r to thwart some data corruption (probabilistic protection)

Diversified Process Replicæ

Optimizations
- Do not simulate FS-related system calls that operate on a FS-objects $\mathcal{O}$ unless $\mathcal{O}$ is shared
- SMP

Dynamic Binary Translation (QEMU)
- $\uparrow$ Does not require program recompilation
- $\uparrow$ Faster then a `ptrace` implementation
- $\downarrow$ Partial overwrite protection is lost

Program Transformation
- Insert non-overlapping gaps between buffer-like variables of P and P_r to thwart some data corruption (probabilistic protection)

Taint-enhanced Anomaly Detection

- Apply the same technique to a broader class of vulnerabilities (e.g., web application vulnerabilities)
- Preliminary results
  - Context-sensitive analysis on a taint-enhanced PHP interpreter
  - Learning policy for SQL injection attacks deals with
    - 2^{nd} order SQL injection on tainted query
    - Dynamic construction of SQL query (e.g., fuzzy advanced search)
  - Leverage on the learning-based approach to learn safe attack pattern usage

Table of Contents

Motivation

Memory Error
- Research Goal

State of the Art
- Artificial Diversity
- Taint Analysis
- Anomaly Detection

Proposed Approaches
- Diversified Process Replicæ
- Taint-enhanced Anomaly Detection

Related Works

Future Directions

Conclusions

Conclusions

- Memory error attacks are still a big threat to software security
- State of the art approaches have drawbacks
  - Mostly *probabilistic* protection
  - Hard to deal with data and data pointer corruption
  - Vulnerable to evasions (e.g., brute forcing, mimicry)
- Diversified process replicæ
  - ↑ Comprehensive & deterministic code/data pointer protection
  - ↓ No arbitrary data corruption protection
- Taint-enhanced anomaly detection
  - ↑ Comprehensive memory error protection
  - ↑ Deterministic code pointer protection
  - ↑ Probabilistic data and data pointer protection
  - ↑ Low false positives rate
  - ↓ It requires a learning-phase

Conclusions

- Memory error attacks are still a big threat to software security
- State of the art approaches have drawbacks
  - Mostly *probabilistic* protection
  - Hard to deal with data and data pointer corruption
  - Vulnerable to evasions (e.g., brute forcing, mimicry)
- Diversified process replicæ
  - $\uparrow$ Comprehensive & deterministic code/data pointer protection
  - $\downarrow$ No arbitrary data corruption protection
- Taint-enhanced anomaly detection
  - $\uparrow$ Comprehensive memory error protection
  - $\uparrow$ Deterministic code pointer protection
  - $\uparrow$ Probabilistic data and data pointer protection
  - $\uparrow$ Low false positives rate
  - $\downarrow$ It requires a learning-phase

Conclusions

- Memory error attacks are still a big threat to software security
- State of the art approaches have drawbacks
  - Mostly *probabilistic* protection
  - Hard to deal with data and data pointer corruption
  - Vulnerable to evasions (e.g., brute forcing, mimicry)
- Diversified process replicæ
  - ↑ Comprehensive & deterministic code/data pointer protection
  - ↓ No arbitrary data corruption protection
- Taint-enhanced anomaly detection
  - ↑ Comprehensive memory error protection
  - ↑ Deterministic code pointer protection
  - ↑ Probabilistic data and data pointer protection
  - ↑ Low false positives rate
  - ↓ It requires a learning-phase

Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar.
Dataflow anomaly detection.
In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 48–62, Washington, DC, USA, 2006. IEEE Computer Society.

Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer.
Defeating Memory Corruption Attacks via Pointer Taintedness Detection.
In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 378–387, Washington, DC, USA, 2005. IEEE Computer Society.

Elena Gabriela Barrantes and David H. Ackley and Stephanie Forrest and Darko Stefanovic.
Randomized instruction set emulation.
ACM Trans. Inf. Syst. Secur., 8(1):3–40, 2005.

H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong.
Anomaly Detection using Call Stack Information.
IEEE Symposium on Security and Privacy, Oakland, California, 2003.

S. A. Hofmeyr, S. Forrest, and A. Somayaji.
Intrusion Detection Using Sequences of System Calls.
Journal of Computer Security, 6(3):151–180, 1998.

Hovav Shacham and Matthew Page and Ben Pfaff and Eu-Jin Goh and Nagendra Modadugu and Dan Boneh.

On the Effectiveness of Address-Space Randomization.
In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307, New York, NY, USA, 2004. ACM Press.

Jingfei Kong, Cliff C. Zou, and Huiyang Zhou.
Improving Software Security via Runtime Instruction-level Taint Checking.
In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24, New York, NY, USA, 2006. ACM Press.

D. Mutz, F. Valeur, C. Kruegel, and G. Vigna.
Anomalous System Call Detection.
ACM Transactions on Information and System Security, 9(1):61–93, February 2006.

Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer.
Exploiting Execution Context for the Detection of Anomalous System Calls.
In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, 2007.

James Newsome and Dawn Xiaodong Song.
Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.
In *NDSS*, 2005.

Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar.
Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.
In *12th USENIX Security Symposium*, 2003.

Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney.
Efficient Techniques for Comprehensive Protection from Memory Error Exploits.
In *14th USENIX Security Symposium*, 2005.

scut / team teso.
Exploiting Format String Vulnerabilities, September 2001.
version 1.2.

R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni.
A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors.
IEEE Symposium on Security and Privacy, Oakland, California, 2001.

The PaX Team.
PaX: Address Space Layout Randomization (ASLR).

http://pax.grsecurity.net.

Wei Xu, Sandeep Bhatkar, and R. Sekar.
Taint-enhanced Policy Enforcement: a Practical Approach to Defeat a Wide Range of Attacks.
In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

Q&A

Thank You!
Q&A?

Backup Material

Backup Material

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
- sync at `mmap` or `msync` time

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
- either P or P_r has to modify the resource R
- they must operate on an *up-to-dated* view of the shared resource R

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 let P to execute a single instruction that accesses the shared area
  - if P has mutual access to R, this is reflected to R and P AS
- T replicates the effect made by P into P_r AS

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