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


- 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


- 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 Replicae
  Taint-enhanced Anomaly Detection
Related Works
Future Directions
Conclusions
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
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
Code pointer corruption

```c
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 Replicas
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
  ⇒ 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

  ⇒ 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
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 \textit{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 $M'$ is consistent with the behavioral profile $M$ learnt during a learning or training phase.

- Deviation from $M$ 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 $M'$ is consistent with the behavioral profile $M$ learnt during a learning or training phase.

- Deviation from $M$ 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.
Motivation
Memory Error
State of the Art
Proposed Approaches
Related Works
Future Directions
Conclusions

Artificial Diversity
Taint Analysis
Anomaly Detection

Anomaly Detection

Determines whether a process behavioral profile $M'$ is consistent with the behavioral profile $M$ learnt during a learning or training phase

- Deviation from $M$ 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

Lorenzo Cavallaro
Comprehensive Memory Error Protection
February, 14 2008
15 / 50
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 Replica\textsubscript{e}

Framework

Idea

To couple the concept of \textit{artificial diversity} and \textit{process replication}

- $T$, the tracer, creates $P_r$, a \textit{replica} of $P$
- $T$ makes $P$ and $P_r$ to behave identically on benign input
- $P$ and $P_r$ are \textit{artificially diversified}

$\Rightarrow$ Detect behavioral divergence caused by malicious input (i.e., memory error attacks)
Process Replication

Rendez-vous

\( T \) 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-vous

\( T \) 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-vous

\( T \) 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., \texttt{brk})
- \( T \) carefully handles other system calls (e.g., \texttt{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

```c
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

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

// 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

<table>
<thead>
<tr>
<th>#</th>
<th>Throughput</th>
<th>MB/s (no DPR)</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>$\sim$ 3.8%</td>
</tr>
<tr>
<td>4</td>
<td>thttpd (read-nocache)</td>
<td>12603.7</td>
<td>7086.3</td>
<td>$\sim$ 43.8%</td>
</tr>
<tr>
<td>5</td>
<td>thttpd (read-nocache-single)</td>
<td>9134.5</td>
<td>2838.1</td>
<td>$\sim$ 69%</td>
</tr>
</tbody>
</table>
## Experimental Results

**Latency Penalties**

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

<table>
<thead>
<tr>
<th>#</th>
<th>Latency</th>
<th>ms (real system)</th>
<th>ms (DPR)</th>
<th>slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>thttpd (mmap)</td>
<td>3.5</td>
<td>4.6</td>
<td>31%</td>
</tr>
<tr>
<td>2</td>
<td>thttpd (mmap-nocache)</td>
<td>3.5</td>
<td>4.5</td>
<td>29%</td>
</tr>
<tr>
<td>3</td>
<td>thttpd (read)</td>
<td>3.5</td>
<td>5.3</td>
<td>51%</td>
</tr>
<tr>
<td>4</td>
<td>thttpd (read-nocache)</td>
<td>3.7</td>
<td>21.6</td>
<td>$\sim 6x$</td>
</tr>
<tr>
<td>5</td>
<td>thttpd (read-nocache-single)</td>
<td>166</td>
<td>646</td>
<td>$\sim 4x$</td>
</tr>
</tbody>
</table>
Table of Contents

Motivation
Memory Error
  Research Goal
State of the Art
  Artificial Diversity
  Taint Analysis
  Anomaly Detection
Proposed Approaches
  Diversified Process Replicae
  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)
    - 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

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

- 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

\[ \Sigma = \{sinks\}, s(a_1, \ldots, a_n) \in \Sigma, a_i \text{ sinks arguments, } i \in \{1, \ldots, n\} \]

- 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

```c
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 ($15,000+ \text{ LoC}$)
  - Python for automatic code generation
Experimental Results I

<table>
<thead>
<tr>
<th>#</th>
<th>App</th>
<th># Traces (Learning)</th>
<th># Traces (Detection)</th>
<th>FP</th>
<th>Overall FPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>proftpd</td>
<td>68,851</td>
<td>983,740</td>
<td>200</td>
<td>2.0 × 10^{-4}</td>
</tr>
<tr>
<td>2</td>
<td>apache</td>
<td>58,868</td>
<td>688,100</td>
<td>2000</td>
<td>2.9 × 10^{-3}</td>
</tr>
</tbody>
</table>

**Table:** Overall False Positives.

<table>
<thead>
<tr>
<th>#</th>
<th>App</th>
<th>Taint</th>
<th>LCP</th>
<th>Min</th>
<th>Struct Inf.</th>
<th>T. Max</th>
<th>Overall FPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>proftpd</td>
<td>3.0 × 10^{-5}</td>
<td>3.0 × 10^{-5}</td>
<td>0</td>
<td>1.4 × 10^{-4}</td>
<td>0</td>
<td>2.0 × 10^{-4}</td>
</tr>
<tr>
<td>2</td>
<td>apache</td>
<td>0</td>
<td>4.3 × 10^{-4}</td>
<td>0</td>
<td>2.4 × 10^{-3}</td>
<td>0</td>
<td>2.9 × 10^{-3}</td>
</tr>
</tbody>
</table>

**Table:** False Positives Breakdown.
### Experimental Results II

<table>
<thead>
<tr>
<th>#</th>
<th>App</th>
<th>Unkn. untaint. traces</th>
<th>Taint. of sinks args</th>
<th>FPs (taint inf.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>proftpd</td>
<td>$2.1 \times 10^{-4}$</td>
<td>$3.0 \times 10^{-5}$</td>
<td>$2.4 \times 10^{-4}$</td>
</tr>
<tr>
<td>2</td>
<td>apache</td>
<td>$4.3 \times 10^{-4}$</td>
<td>0</td>
<td>$4.3 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

**Table: Unknown/Untainted Traces.**

<table>
<thead>
<tr>
<th>#</th>
<th>App</th>
<th>slowdown (taint)</th>
<th>slowdown (taint-learn)</th>
<th>slowdown (taint-detect)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>proftpd</td>
<td>3.1%</td>
<td>5.9%</td>
<td>9.3%</td>
</tr>
<tr>
<td>2</td>
<td>apache</td>
<td>5.7%</td>
<td>10.3%</td>
<td>18.5%</td>
</tr>
</tbody>
</table>

**Table: Throughput Slowdown.**
Motivation
Memory Error
State of the Art
Proposed Approaches
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 Replicaæ

Optimizations

- Do not simulate FS-related system calls that operate on a FS-objects $O$ unless $O$ is shared
- SMP

Dynamic Binary Translation (QEMU) does not require program recompilation
Faster than a ptrace implementation
Partial overwrite protection is lost

Program Transformation

- Insert non-overlapping gaps between buffer-like variables of $P$ and $Pr$ to thwart some data corruption (probabilistic protection)
Diversified Process Replicae

Optimizations

- Do not simulate FS-related system calls that operate on a FS-objects $O$ unless $O$ is shared
- SMP

Dynamic Binary Translation (QEMU)  
Does not require program recompilation
Faster than 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 $O$ unless $O$ is shared
- SMP

Dynamic Binary Translation (QEMU)
- Does not require program recompilation
- Faster than 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)
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\textsuperscript{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 Replicae
  - 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 replicae
  - 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 replicae
  - 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
  - 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


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.

Jingfei Kong, Cliff C. Zou, and Huiyang Zhou.
Improving Software Security via Runtime Instruction-level Taint Checking.

Anomalous System Call Detection.
Exploiting Execution Context for the Detection of Anomalous System Calls.  

James Newsome and Dawn Xiaodong Song.  
Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation 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.  


The PaX Team. PaX: Address Space Layout Randomization (ASLR).
http://pax.grsecurity.net.

Thank You!
Q&A?
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 $Pr$ create a readable and writable non-anonymous shared memory segment $M$
- $ptr[0]$ points to the beginning of $M$

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