Retrospective Protection utilizing Binary Rewriting

Sergej Proskurin, Fatih Kilic, and Claudia Eckert
{proskuri,kilic,eckert}@sec.in.tum.de

Chair for IT-Security
Technische Universität München
Munich, Germany

20.05.2015
Introduction

**Object code** most often the only form available
- Likely to contain vulnerabilities

**Buffer overflow vulnerabilities** present a common threat
- Insufficient array bounds checks
- Lack of compile-time security hardening measures
  - Performance often favored over security
Object code most often the only form available
- Likely to contain vulnerabilities

Buffer overflow vulnerabilities present a common threat
- Insufficient array bounds checks
- Lack of compile-time security hardening measures
  - Performance often favored over security

Idea: Retrospective binary protection – BinProtect
Introduction

Objectives:

- Mitigation of potential buffer overflows
- Incorporation of protection mechanisms as provided by modern compilers
- Timely decoupled development and security hardening process
Common **compile-time protection mechanisms**: 

- Standard C library consolidation
- Stack protection
- Execution prevention on the stack
- Global Offset Table protection
Common **compile-time protection mechanisms:**

- Standard C library consolidation
- Stack protection
- Execution prevention on the stack
- Global Offset Table protection

**BinProtect** integrates these mechanisms into binaries through:

- Binary Editing - aka post link-time object code transformation
- Transformation of the *Executable and Linking Format* (ELF)
BinProtect utilizes capabilities provided by Dyninst [1] and PatchAPI [2]

- **Dyninst**: Static analysis and instrumentation framework
  - Comprehensive static analysis capabilities
  - Precise deduction of function and control flow

- **PatchAPI**: Binary code patching library
  - Fine granular object code transformation
  - Introduces *Structured Binary Editing* [2]
    - Binaries considered as flow of basic blocks: *Control Flow Graph* (CFG)
    - Binary transformations performed on the CFG level
The standard C library provides a set of unsafe functions:

- `strcpy()`
- `gets()`
- `sprintf()`
- ...

**Issue:** No array bounds checks
Common **protection mechanisms**:  
- **Libsafe [3]:** Dynamically loadable library  
  ▶ Utilizes environment variable *LD_PRELOAD*: Loaded first  
  ▶ Intercepts calls only to *dynamically* linked libc  

- **GCC `-D_FORTIFY_SOURCE`**  
  ▶ Inserts array bounds checks in critical functions  
  ▶ Hardens calls to *statically* or *dynamically* linked libc
BinProtect provides the best of both worlds:

- Interception of calls to **statically linked** libc:
  - Wrapper code assigned to a new ELF segment
  - Trampolines to access wrappers
**BinProtect provides the best of both worlds:**

- Interception of calls to **statically linked** libc:
  - Wrapper code assigned to a new ELF segment
  - Trampolines to access wrappers

- Interception of calls to **dynamically linked** libc:
  - Insert dependency to custom library with safe wrapper implementations
  - Load wrapper library before libc
  - Keep track of original function addresses
  - No need for environment variables
Two types of stack-based buffer overflow protection mechanisms:

1. Buffer overflow *prevention* based on bounds checking
   - Run-time overhead and potential compatibility issues
   - **Not very practical**

2. Buffer overflow *detection* based on integrity checking
   - **Terminator canaries** in stack frames
   - **Shadow stack** for return addresses
Two types of stack-based buffer overflow protection mechanisms:

1. Buffer overflow *prevention* based on bounds checking
   - Run-time overhead and potential compatibility issues
   - *Not very practical*

2. Buffer overflow *detection* based on integrity checking
   - *Terminator canaries* in stack frames
   - *Shadow stack* for return addresses

Common *protection mechanism*:
- GCC `-fstack-protector-(all|strong)` family
  - Implements a canary-based approach
BinProtect incorporates a shadow stack mechanism

- Basic block transformation
  - Localize basic blocks associated with the function’s entry/exit points
  - Encapsulate prologue/epilogue information

- Code injection
  - Inject code representing a new prologue/epilogue
  - Adopt control flow accordingly
Global Offset Table (GOT):

- Holds addresses of *Dynamically Shared Objects* (DSO)
- At compile-time, addresses of DSOs unknown
  - DSO addresses to be relocated at *load-* or *run-time*
Global Offset Table (GOT):

- Holds addresses of *Dynamically Shared Objects* (DSO)
- At compile-time, addresses of DSOs unknown
  - DSO addresses to be relocated at *load-* or *run-time*

The GOT is distributed across two ELF sections:

- `.got`: references to DSOs relocated at *load-time*
- `.got.plt`: references to DSOs relocated at *run-time* *(Lazy Binding)*
Global Offset Table (GOT):

- Holds addresses of *Dynamically Shared Objects* (DSO)
- At compile-time, addresses of DSOs unknown
  - DSO addresses to be relocated at *load-* or *run-time*

The GOT is distributed across two ELF sections:

- `.got`: references to DSOs relocated at *load-time*
- `.got.plt`: references to DSOs relocated at *run-time* (*Lazy Binding*)

**Issue:** Potential manipulation of GOT entries
ELF Transformation

- Global Offset Table Protection

Common **protection mechanism**: *REL*ocation *Read-Only* (RELRO)

- GCC `-Wl,-z,relro (-z,now)`
  - Marks `.got` as read-only after *load-time* relocation

**Partial vs. full RELRO:**

- **Partial RELRO**: *Lazy Binding*
  - `.got.plt` remains writable and hence vulnerable
- **Full RELRO**: Resolves all DSOs at load-time
  - GOT entries cannot be manipulated
  - Trades off startup time for security
ELF Transformation

- Global Offset Table Protection

BinProtect ensures protection of the entire GOT

- Deactivation of *Lazy Binding*
  - Insert tags `DT_BIND_NOW` and `DT_FLAGS_1` into `.dynamic` section

- Relocation of `.got.plt`
  - `PT_GNU_RELRO` describes memory to be marked as read-only
  - Only relocations preceding page boundary to `.data` marked as read-only
    → Relocate `.got.plt` to a dedicated segment

- Relocation read-only
  - Extend `_init()` to change permissions of `.got.plt` before `main()`
Evaluation

Protection of three archiving utilities

- Negligible increase in average execution time
- Noticeable increase in code size due to binary transformations
  - Trampoline based code constructs inserted into original code
  - Original code remains in the binary and thus blows up its size

<table>
<thead>
<tr>
<th>Binary</th>
<th>Hardened functions</th>
<th>Size increase</th>
<th>Time increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU tar, v1.27.1</td>
<td>807 (of 807)</td>
<td>38.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>Gzip, v1.6</td>
<td>67 (of 67)</td>
<td>86.3%</td>
<td>2.3%</td>
</tr>
<tr>
<td>Bzip2, v1.0.6</td>
<td>36 (of 36)</td>
<td>137.5%</td>
<td>3.5%</td>
</tr>
</tbody>
</table>
Conclusion

- BinProtect mitigates potential buffer overflow vulnerabilities
- Performs direct binary editing and ELF transformations
- Results show effective protection with negligible time overhead
ELF defines a common **object file format** in Unix-based systems

- Determines how binaries are handled at load- and run-time
- ELF header acts as a guide to locate essential file components
Writable ELF segments are potential vulnerabilities

- Not always practical to prohibit execution
  - Dynamic code generation
  - Nested functions and signal handling in Linux

- **No eXecute** (NX) bit prevents execution on a per page basis

**Common protection mechanism:**

- GCC `-z noexecstack`
  - Inserts the program header `PT_GNU_STACK` into the ELF file
    → Program loader marks the affected pages as non-executable

- BinProtect incorporates `PT_GNU_STACK` into ELF files
