Securing software by enforcing data-flow integrity

Manuel Costa

Joint work with:
Miguel Castro, Tim Harris

Microsoft Research Cambridge
University of Cambridge
Software is vulnerable

- use of unsafe languages is prevalent
  - most “packaged” software written in C/C++
- many software defects
  - buffer overflows, format strings, double frees
- many ways to exploit defects
  - corrupt control-data: stack, function pointers
  - corrupt non-control-data: function arguments, security variables

defects are routinely exploited
Approaches to securing software

• remove/avoid all defects is hard
• prevent control-data exploits
  – protect specific control-data StackGuard, PointGuard
  – detect control-flow anomalies Program Shepherding, CFI
  – attacks can succeed without corrupting control-flow
• prevent non-control-data exploits
  – bounds checking on all pointer dereferences CRED
  – detect unsafe uses of network data Vigilante, [Suh04], Minos, TaintCheck, [Chen05], Argos, [Ho06]
  – expensive in software

no good solutions to prevent non-control-data exploits
Data-flow integrity enforcement

• compute data-flow in the program statically
  – for every load, compute the set of stores that may produce the loaded data
• enforce data-flow at runtime
  – when loading data, check that it came from an allowed store
• optimize enforcement with static analysis
Data-flow integrity: advantages

• broad coverage
  – detects control-data and non-control-data attacks

• automatic
  – extracts policy from unmodified programs

• no false positives
  – only detects real errors (malicious or not)

• good performance
  – low runtime overhead
Outline

• data-flow integrity enforcement
• optimizations
• results
Data-flow integrity

• at compile time, compute reaching definitions
  – assign an id to every store instruction
  – assign a set of allowed source ids to every load

• at runtime, check actual definition that reaches a load
  – runtime definitions table (RDT) records id of last store to each address
  – on store(value,address): set RDT[address] to store’s id
  – on load(address): check if RDT[address] is one of the allowed source ids

• protect RDT with software-based fault isolation
Example vulnerable program

```c
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);
```

- non-control-data attack
- very similar to a real attack on a SSH server
Static analysis

• computes data flows conservatively
  – flow-sensitive intraprocedural analysis
  – flow-insensitive interprocedural analysis
    • uses Andersen’s points-to algorithm
    • scales to very large programs

• same assumptions as analysis for optimization
  – pointer arithmetic cannot navigate between independent objects
  – these are the assumptions that attacks violate
SETDEF authenticated 1
int authenticated = 0;
char packet[1000];

while (CHECKDEF authenticated in {1,8} !authenticated) {
    PacketRead(packet);

    if (Authenticate(packet)) {
        SETDEF authenticated 8
        authenticated = 1;
    }
}

CHECKDEF authenticated in {1,8}
if (authenticated)
    ProcessPacket(packet);
Runtime: detecting the attack

Vulnerable program

```c
SETDEF authenticated 1
int authenticated = 0;
char packet[1000];

while (CHECKDEF authenticated in {1,8}!
authenticated) {
    PacketRead(packet);

    if (Authenticate(packet)) {
        SETDEF authenticated 8
        authenticated = 1;
    }
}

CHECKDEF authenticated in {1,8}
if (authenticated)
    ProcessPacket(packet);
```

Memory layout

RDT slot for authenticated
stores disallowed above 0x40000000

authenticated stored here

Attack detected!
definition 7 not in {1,8}
Also prevents control-data attacks

• user-visible control-data (function pointers,…)  
  – handled as any other data
• compiler-generated control-data  
  – instrument definitions and uses of this new data  
  – e.g., enforce that the definition reaching a \texttt{ret} is generated by the corresponding \texttt{call}
Efficient instrumentation: SETDEF

• SETDEF \_authenticated 1 is compiled to:

```
lea ecx,[_authenticated]
test ecx,0C0000000h
je L
int 3
L:
shr ecx,2
mov word ptr [ecx*2+40001000h],1
```

- Get address of variable
- Prevent RDT tampering
- Set RDT[address] to 1
Efficient instrumentation: CHECKDEF

CHECKDEF _authenticated \{1,8\} is compiled to:

```
lea ecx,[_authenticated]
shr ecx,2
mov cx, word ptr [ecx*2+40001000h]
cmp cx, 1
je L
cmp cx, 8
je L
int 3
L:  ```

- Get address of variable
- Get definition id from RDT[address]
- Check definition in \{1,8\}
Optimization: renaming definitions

• definitions with the same set of uses share one id

```c
SETDEF authenticated 1
int authenticated = 0;
char packet[1000];
while (CHECKDEF authenticated in {1,8} !authenticated) {
    PacketRead(packet);
    if (Authenticate(packet)) {
        SETDEF authenticated & authenticated = 1;
    }
}
CHECKDEF authenticated in {1,8}
if (authenticated)
    ProcessPacket(packet);
```
Other optimizations

• removing SETDEFs and CHECKDEFs
  – eliminate CHECKDEFs that always succeed
  – eliminate redundant SETDEFs
  – uses static analysis, but does not rely on any assumptions that may be violated by attacks

• remove bounds checks on safe writes

• optimize set membership checks
  – check consecutive ids using a single comparison
Evaluation

• overhead on SPEC CPU and Web benchmarks
• contributions of optimizations
• ability to prevent attacks on real programs
Runtime overhead

The bar chart shows the normalized execution time for different benchmarks: gzip, vpr, mcf, crafty, bzip2, twolf, art, equake, and ammp. The chart compares the base time with intraproc DFI and interproc DFI.
Contribution of optimizations

normalized execution time

without renaming
with renaming
with all optimizations

gzip, vpr, mcf, crafty, bzip2, twolf, art, equake, ammp
Overhead on SPEC Web

maximum overhead of 23%
## Preventing real attacks

<table>
<thead>
<tr>
<th>Application</th>
<th>Vulnerability</th>
<th>Exploit</th>
<th>Detected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>NullHttpd</td>
<td>heap-based buffer overflow</td>
<td>overwrite cgi-bin configuration data</td>
<td>yes</td>
</tr>
<tr>
<td>SSH</td>
<td>integer overflow and heap-based buffer overflow</td>
<td>overwrite authenticated variable</td>
<td>yes</td>
</tr>
<tr>
<td>STunnel</td>
<td>format string</td>
<td>overwrite return address</td>
<td>yes</td>
</tr>
<tr>
<td>Ghttpd</td>
<td>stack-based buffer overflow</td>
<td>overwrite return address</td>
<td>yes</td>
</tr>
</tbody>
</table>
Conclusion

- enforcing data-flow integrity protects software from attacks
  - handles non-control-data and control-data attacks
  - works with unmodified C/C++ programs
  - no false positives
  - low runtime and memory overhead
Overhead breakdown

% contribution to overhead

store id  bounds check  load id  compare id
Contribution of optimizations