CS-527 Software Security

Practical Defenses

Asst. Prof. Mathias Payer

Department of Computer Science
Purdue University

TA: Kyriakos Ispoglou

https://nebelwelt.net/teaching/17-527-SoftSec/

Spring 2017

Table of Contents

Bug Fixing as an Insider

Assume you work for $BIGCOMPANY and you have found a severe security bug in the code of one of the products. What do you do?

- You report the vulnerability to the security team.
- Together with the security team you will coordinate the development of the fix for the vulnerability and devise how to update the software.
- You devise a plan to update the software that is used in the wild.

Bug Fixing as an Outsider

Assume you do not work for the company and you have found a security bug. What do you do?

- You have several options: report it to the company (responsible disclosure), you announce it openly (full disclosure), you stockpile the vulnerability, you sell it to the highest bidder, or you exploit the vulnerability yourself. Depending on the country you life in, the last two options are likely illegal.
- Let's assume you take the high road (responsible disclosure).
- You report the vulnerability to the security team and establish a time window when you will publicly release the vulnerability. (Note that for many of the bug bounty programs you are not allowed to publicly release the vulnerability.)
- ...

Bug Fixing as an Outsider

Assume you do not work for the company and you have found a security bug. What do you do?

- .. you will not hear back for days, weeks, or months as the security team coordinates the internal bug fixing process.
- At one point you will hear back on how they handled "your" vulnerability.
- Orthogonally you can use a mediator like MITRE who will also assign a CVE number (Common Vulnerability and Exposures) to your vulnerability.

The Update and Patching Process

- How do you distribute updates to your costumers?
- You may send out disks/CDs for new versions.
- You can provide the new releases on a website (e.g., most open source software and Linux distributions use web sites)
- Many software products nowadays have an automatic update service and the software (or at least the automatic updater) periodically polls for new updates.
- New software patches can be delivered on regular intervals ("patch Tuesday"), out-of order when an emergency update is deemed to be required, or whenever a patch is available.

Software Updating

- Software updating is an interesting and active research area that addresses several problems.
- How can you deliver patches more efficiently: binary delta patching (e.g., Google Chrome) and rolling releases.
- How can you distribute the load for software updates across millions of users? (Assume you have limited server capacity.)
- How do you update a running software component without restart? (Otherwise, when do you update?)

Table of Contents

Control-Flow Integrity (CFI)

CFI Definition

CFI is a defense mechanism that protects applications against control-flow hijack attacks. A successful CFI mechanism ensures that the control-flow of the application never leaves the predetermined, valid control-flow that is defined at the source code/application level. This means that an attacker cannot redirect control-flow to alternate or new locations.

Sketching an implementation

- Each CFI implementation consists of a (static) analysis phase that constructs a control flow graph and a dynamic enforcement phase that restricts control-flow transfers.
- The static analysis can be implemented through a simple static points-to analysis that, for each indirect control flow transfer location in the source code determines the set of targets it may point to.
- The policy enforcement can be implemented through either a set check or through ID classes.
- (Note that ID classes will increase the imprecision due to coalescing to single ID classes.)

Sketching an implementation

- The original CFI proposal used a simple static analysis and ID classes while later proposals mostly shifted towards set checks.
- An interesting diversion was coarse-grained CFI that reduced the amount of target classes to two or three (for function returns, function calls, and indirect jumps).
- Challenges for CFI implementations are modularity and source compatibility.

Limitation of CFI

- CFI allows the underlying bug to fire and the memory corruption can be controlled by the attacker. The defense only detects the deviation after the fact, i.e., when a corrupted pointer is used in the program.
- What kind of attacks are possible?
- An attacker is free to modify the outcome of any JCC
- An attacker can choose any allowed target at each ICF location
- For return instructions: one set of return targets is too broad and even localized return sets are too broad for most cases.
- For indirect calls and jumps, attacks like COOP (Counterfeit Object Oriented Programming) have shown that full functions can be used as gadgets.

Table of Contents

Setting

- Memory corruption is abundant.
- Strong memory-safety-based defenses have not been adopted.
- Weaker defenses like strong memory allocators also ignored.
- Only defenses that have *very low* overhead are adapted.
- What if we can have memory safety but only where it matters?
- Assume we want to protect applications against control-flow hijacking attacks. What data must be protected?
- Code Pointer Integrity (CPI) ensures that all code pointers are protected at all times.

Attacker model

- Attacker can read data, code
  (includes stack, bss, data, text, heap).
- Attacker can write data.
- Attacker cannot modify code.
- Attacker cannot influence the loading process.

Quiz: what is a control-flow hijack attack?

- What is the difference between a data-only attack and a control-flow hijack attack?
- For control-flow hijack attacks, the attacker captures and redirects control-flow to an attacker-controlled location (that would otherwise not be reached by a benign program execution).

Existing memory safety solutions

- SoftBound+CETS 116% overhead, only partial support for SPEC CPU2006
- CCured: 56% overhead
- AddressSanitizer: 73% overhead, only partial memory safety (probabilistic spatial).

How to enforce memory safety?

```
1  char *buf = malloc(10);
2  buf_lo = p; buf_up = p+10;
3  ...
4  char *q = buf + input;
5  q_lo = buf_lo; q_up = buf_up;
6  if (q < q_lo || q >= q_up)
7   abort();
8  *q = input2;
9  ...
10 (*func_ptr)();
```

A paradigm shift: protect select data

Protecting select data

Instead of protecting everything a little protect a little completely.
Strong protection for a select subset of data. Attacker may modify any unprotected data.
By only protecting code pointers, CPI reduces the overhead of memory safety from 116% to 8.4% while still deterministically protecting applications against control-flow hijack attacks.

What data must be protected?

- Sensitive pointers are code pointers and pointers used to access sensitive pointers.
- We can over-approximate and identify sensitive pointer through their types: all types of sensitive pointers are sensitive.
- Over approximation only affects performance.

Memory layout

- The regular memory view is split into two views: control plane and data plane.
- The control plane is a view that only contains code pointers (and transitively all related pointers).
- The data plane contains only data, code pointers are left empty (void/unused data).
- As a software security specialist, what do you ask?
- The two planes must be separated and data in the control plane must be protected from pointer dereferences in the data plane.

Safe stack

- So far we covered the heap, but what about the stack?
- Run a compiler analysis on each stack frame, push any unsafe variable to an unsafe stack in the data plane, keep all safe variables in the control plane.
- How can we determine if a variable is safe?
- Anything that escapes the stack frame is unsafe. Any unsafe pointer arithmetic makes the variable unsafe.

Implementation

- Existing implementation builds on LLVM, protects FreeBSD and a large set of applications.
- Low overhead in practice: 8.4% for SPEC CPU2006 and 1.9% for a weaker policy.
- Key insight: memory safety for code pointers only.

Table of Contents

Summary

- Control-Flow Integrity (CFI) allows the memory safety error to happen but restricts the attacker to choose targets for indirect control flow transfers from a well-defined set.
- CFI is a practical defense with low performance overhead.
- CPI is a stronger defense that separates code pointers and sensitive types from the attacker. An attacker can no longer corrupt a code pointer.
- SafeStack is now available in the LLVM sanitizer framework.

Questions?

?