Shadowclone: Thwarting and Detecting DOP Attacks with Stack Layout Randomization and Canary

EECS 583
Yunjie Pan, Shibo Chen, Cheng Chi, Yifan Guan
Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo
Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo
DOP Example

Modeled after FTP server

Uses a stack buffer overflow vulnerability to control a few stack variables

```c
struct server { int *cur_max, total, typ; } *srv;

int connect_limit = MAXCONN; int *size, *type;

char buf[MAXLEN];

size = &buf[8]; type = &buf[12];
...

while (connect_limit--) {
    readData(sockfd, buf);  // stack bof
    if (*type == NONE) break;
    if (*type == STREAM) {
        *size = *(srv->cur_max);  // dereference
    } else {
        srv->typ = *type;  // assignment
        srv->total += *size;  // addition
    }
}
...
```
Simple Example

Simple Stack Overflow
If successful, var1 and var2 will be changed to 583
Prior work - Smokestack

Randomizes the order of stack variables during runtime with P-BOX

+ Much harder to deliver DOP attacks

+ Negligible memory overhead

- Runtime performance overhead

- Cannot detect attacks when happening
Goals

- Reduce runtime overhead by compile time randomization
- Detect attacks when happening
Threat Model

- CFI (Control Flow Integrity) defenses deployed
- Stack buffer overflow vulnerability
- Attackers cannot see the code, but can learn gradually
Shadowclone

● Motivation & Background

● Methodology

● Implementation

● Evaluation

● Demo
Shadowclone

- Generate compile-time randomized clones of vulnerable functions
- Insert compile-time random canary into stack variables and check before the function returns
- Randomly select copy to execute in run time
#include <stdio.h>
#include <stdlib.h>

void func(){
  double var1 = 483.0;
  int var2 = 483;
  char var3 = 'e';
  char* buff_ptr = &buff[0];
  size_t size = 1024;
  getline(&buff_ptr, &size, stdin);
  printf("var1: %f\n", var1);
  printf("var2: %i\n", var2);
  printf("var3: %c\n", var3);
  printf("buff: %s\n", buff);
  printf("This is function func.\n");
}

int main(){
  printf("Calling func:\n");
  func();
  printf("func returned.\n");
}

void func0(){
  int var2 = 483;
  uint32_t canary = 1092384;
  double var1 = 483.0;
  char buff[4];
  char var3 = 'e';
  char* buff_ptr = &buff[0];
  size_t size = 1024;
  getline(&buff_ptr, &size, stdin);
  if (canary != 1092384){
    exit(1);
  }
}

void func1(){
  void *fp_index = rand() % 3;
  if (fp_index == 0){
    func0();
  } else if (fp_index == 1){
    func1();
  } else {
    func2();
  }
}

void func2(){
  // call function
}
Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo
Implementation

1. Find all concrete functions (except main and syscall)
2. Find all alloca instructions
3. Clone a function \(\text{min}(threshold, \text{num(alloca)!})\) times
4. Randomize the order of stack variables
5. Insert the canary and checks
6. Convert original function to randomly select a clone in run-time
Randomize the order of stack variables

Generate a random ordering (a configuration)

If this configuration already exists:
    Continue
Else:
    Apply this config to one of the clones

Repeat until all clones have been randomized
Insert canary and checks

1. Randomly select an insertion point and insert a 32-bit canary
2. Generate a random number and store it to the location of our canary
3. Insert a *compare-and-branch* duo before each *return* instruction
   (branch to the exception handler if compromise detected)
```c
1 define void @func( ) #0 { 
2 %1 = alloca double, align 8
3 %2 = alloca i32, align 4
4 %3 = alloca i8, align 1
5 %4 = alloca [4 x i8], align 1
6 %5 = alloca i8*, align 8
7 %6 = alloca i64, align 8
8 store double 4.830000e+02, double* %canary
9 store i32 483, i32* %canary
10 store i8 101, i8* %3
11 %7 = getelementptr inbounds [4 x i8], [4 x i8]* %var1, i64 0
12 store i8* %7, i8** %var2
13 store i64 1024, i64* %6
14 %8 = load %struct._IO_FILE*, %struct._IO_FILE** %buff_ptr
15 %9 = call i64 @getline(i8** %buff_ptr, i64* %size)
16 %1 = load i32, i32* __sFILE* %9
17 %2 = icmp eq i32 %1, 780689205
18 br i1 %2
19 ret void
```

Canary Var

Hard Coded CMP
Run-time Selection

get_rand() is defined in our run-time library

Generates a 32-bit random number with the RDRAND instruction

```c
1 define dso_local void @func() #0 {
2   rand_bb:
3       %0 = call i32 @get_rand()
4       %1 = icmp eq i32 %0, 0
5       br i1 %1, label %func_func.1, label %ctrl0
6
7   func_func.1:
8       call void @func.1()
9       ret void
10
11   func_func.2:
12       call void @func.2()
13       ret void
14
15   func_func.3:
16       call void @func.3()
17       ret void
18
19   func_func.4:
20       call void @func.4()
21       ret void
22
23   ctrl0:
24       %2 = icmp eq i32 %0, 1
25       br i1 %2, label %func_func.2, label %ctrl1
26
27   ctrl1:
28       %3 = icmp eq i32 %0, 2
29       br i1 %3, label %func_func.3, label %func_func.4
30 }
```
Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo
Experiment Setup

**Platform:**

Xeon Gold 6126, Ubuntu 18.04 Linux, 256GB of memory

**Benchmarks:**

Three in-House testcases (*big_array*, *wc*, and *compress*)

Three Spec06 benchmarks (*bzip2*, *mcf*, and *h264ref*)

**Source of random numbers:**

RDRAND
Performance Overhead

- Variance of 4
- Variance of 8
- Variance of 16

- wc: -5.13%
- compress: 156.29%
- Big Array: 156.73%
- mcf
- bzip2: 101.76%
- h264ref: 639.20%
Spatial Locality: *Code Size (in KB)*
Spatial Locality: # of L-Cache Misses

- Variance of 4
- Variance of 8
- Variance of 16
Temporal Locality & Speculation: # of Branch Mispredictions

![Graph showing branch mispredictions and performance overhead for different benchmarks and variance classes.](image-url)
Security Analysis

- The attacker learns quickly
  - Learns about any configuration after this very configuration has been run only once
- The attacker doesn’t trigger any exception by accident

Metrics:

What’s the chance for an attacker to successfully compromise our system without being detected?
Security Analysis

Probability of Attackers Successfully Deliver Attack w/o Being Detected

- Compromise 1 function
- Compromise 2 functions
- Compromise 3 functions

Variance of 4
Variance of 8
Variance of 16
Shadowclone

- Motivation & Background
- Methodology
- Implementation
- Evaluation
- Demo
Conclusion

• Shadowclone can efficiently thwarts and detects DOP attacks.

• Shadowclone has low performance overhead when running small programs. Its performance deteriorates as the size of program gets larger and the program gets more function calls.
Question?
Probability of Attackers Successfully Deliver Attack w/o Being Detected

- \( P(\text{attacker succeeds without being attacked}) = \sum_{k=1}^{\infty} P(\text{the first } (k-1) \text{ times failed and without being attacked}) \times P(\text{the } k\text{th time succeeds and not being detected}) \)

- \( P(\text{the first time succeed}) = 1/N! \quad P(\text{the } k\text{th time succeed}) = 1/M \)
  - (N is the average number of stack variables in a function, M is the number of clones)
  - N = 10 in the benchmarks we analyzed

- \( P(\text{an attack would be detected}) = 1/2 \)