Patch unsafe input calls at binary level

MyeongGeun Shin
mg.shin@kaist.ac.kr
KAIST

Abstract

There is a competition named K-cyber security challenge (https://www.k-csc2020.com/). This competition is also known as KCGC which named after DARPA's Cyber Grand Challenge. For KCGC 2020, competition consists of two parts. One is automatic exploit generation, and another one is automatic patcher. Patching in binary level is interesting research area as well as finding vulnerabilities. So in this project, I will propose some formal methods and heuristics to solve this problem.

Keywords: static analysis, binary patch

1 Motivating example

Let's dive into real examples.

```
#include <stdio.h>

void shell(){ system("cat flag"); }

int main(){
    char buf[0x20];
    gets(buf);
    return 0;
}
```

Let's assume that Canary mitigation is off. The code above has vulnerability which is CWE-121: Stack-based Buffer Overflow. That code will likely to generate corresponding assembly code like below.

```
push ebp
mov ebp, esp
sub esp, 0x30
; Other commands
lea edx, [ebp-0x2c]
push edx
call gets
; Other commands
leave
ret
```

We can observe that fuction `gets` takes `[ebp-0x2c]` as it's argument. And `gets` doesn't limit input stream so this is vulnerable point of this program. And we know that any part of program will write no more over `ebp`. So, we can sanitize that command into `fgets(buf, 0x30, stdin)`.

If function has other variables, and use some variable at some point, we can mark all these use-points, and can limit buffer length to next use-point.

And, for `malloc`ed chunks. If we can find out the point where it has been allocated, we can patch the input function with dedicated input length.

2 Implementation

2.1 Make a patch

We can instrument binary to have another section for patch only. And then, substitute patching location with `jmp patch_code` and at `patch_code` we write original code as well as sanitizing part. At last part of `patch_code` there will be another jump that jump backs into original part of the program.

2.2 Size modeling

I will find patterns like this

- `mov [using_part], other value`
- `lea general_registers, [using_part]`
  `push general_registers`

and mark them as **Use-point** of that function. For chunks in stack, let's define the size as the length to the next use-point.

And, allocated chunk will have it's allocation size as it's chunk size.

2.3 Modification

Original	Modification
`scanf("\%s")`	`scanf("\%[size]s")`
`gets(buf)`	`fgets(buf,size,stdin)`
`read(0,buf,big_size)`	`read(0,buf,size)`
Etc…	

And of course, we can also trim output functions as well. That patch will significantly reduce risk of information leak vulnerabilities. (e.g. patch `printf("\%s",buf)` into `printf("\%.[length]s",buf)`

3 Discussion

This approach lacks accuracy in many aspects, but it definitely have some meanings especially in KCGC. And I think it's a good start to play with toy language. I will gradually increase the coverage and accuracy of this proposal.