18739L Research - Taint Analysis for CTF problems

Rajiv Kulkarni

May 3, 2015

1 Introduction

Many problems in CTFs involve hijacking the control flow during the execution of a program. This includes the likes of stack based buffer overflows, heap overflows and format string vulnerabilities. The exploit involves providing user input such that the control-flow of the program is manipulated to perform unintended actions. All the stack based buffer overflow problems where one has to overwrite the return address and execute a shell code fall under this category.

I have written a tool to identify which of the elements of a program can be controlled by an attacker. The tool helps in identifying all the locations in the memory space where the user-input can be percolated. It identifies if the eip register can be controlled (which means that the execution flow can be hijacked). In addition to that, the tool also identifies which registers can be controlled as well as what memory regions in the stack, heap and global space.

This tool assists the attacker in efficiently detecting stack overflows and also other vulnerabilities such as heap overflows, off-by-one errors, use-after-free errors and other miscellaneous vulnerabilities which depend on how a malicious user input interacts with the program. The attacker will be able to see what values are being affected directly or indirectly by the input and thus develop an appropriate strategy to exploit a given program.

This tool would be able to identify few values like return addresses, function arguments, etc but otherwise would just provide the memory addresses and registers an attacker can control. It is then the attacker's decision on how to exploit the program based on the values under control.

2 Approach

This tool infers the attacker controlled regions in memory using dynamic taint analysis. A user input of any form is considered tainted, and this tool dynamically instruments the given binary and propagates the taint accordingly. The tainted regions include user inputs, command line arguments, environment variables, network inputs, file inputs, etc. However, in this proof-of-concept, only the data read from a file is tainted.

The following code provides an example of what values can be tainted. Here, 32 bytes of buffer is allocated using malloc and data is read into it from a file. Hence, at this point, the whole of 32 bytes must be tainted. In 'fun1', few values of 'buf' are modified (e.g. buf[2] = 9). These values are now n longer controlled by the attacker and hence the taint should be removed from such memory locations.

```
int fun1(char *buf)
{
  char a,b,c;

  a = buf[8];
  b = 10;
  c = buf[0];
  buf[6] = b;
  buf[2] = 9;
  buf[4] = a;
  buf[9] = 0;
  return 0;
}

int main(int ac, char **av)
{
  int fd;
  char *buf;

  if (!(buf = malloc(32)))
    return -1;

  fd = open("./file.txt", O_RDONLY);
  read(fd, buf, 32), close(fd);
  fun1(buf);
}
```

The main idea of this tool is to perform taint analysis on the input during run-time. The tool aims to analyze the binary for calls such as fgets, scanf ,etc which take input from the user. Also, the tool aims to check for any command line arguments and also environment variables and calls to fread. In short, the tool gathers all the places through which a malicious input can creep into the program memory space. It then marks those elements as "tainted". For now, the tool instruments only "read" system call. In the above code, the user input is read into "buf". Hence, the chunk into which the user-input is read (need not be limited to sizeof(buf)) is marked tainted. This taint is then propagated further down the execution path of the program. The attacker can monitor the execution of the program and also the "taint-state" that is maintained by the tool, to determine exactly what elements can an attacker control and when. This would help the attacker immensely in deciding how to exploit the given program.

Apart from user-input taint tracking, one more major feature of this tool is to track the heap. This is useful where there are heap based vulnerabilities like heap overflows and use-after-free bugs. An attacker can immediately identify that a buffer on the heap is in tainted state even after it is freed.

The tool also gives which registers can be controlled by the attacker. This information would be useful during ASLR problems where attacker is usually looking for a constant reference and a constant offset such as difference between EAX and current stack pointer.

3 Implementation

My initial plan was to use BAP to do binary instrumentation and use its taint tracking module to do the analysis. The first step involved generating execution traces for a binary. I planned to achieve that using intel pintools. The next step was to use the "Traces_backtaint" module on those traces. However, this approach was not ideal since the Traces_backtaint module was removed from the BAP source.

I then created a pintool which instrumented the binary at instruction level. I used the same tool to maintain the "taint state" and track tainted memory. Intel pintools allows various callback functions for different types of events. The two main callback functions used in this tool were "PIN_AddSyscallEntryFunction" and "INS_AddInstrumentFunction". These two register callbacks when a system call is made and when each instruction is executed respectively.

I am catching read system call and its arguments. Assuming that all data read would be tinted, the tool marks the memory location into which the data is read as tainted. It also has the number of bytes read . Hence, the next corresponding bytes in the memory are also tainted. This information is stored in a global list. From here on, any memory region that involves the use of this tainted region would be marked tainted. For e.g., if this memory is assigned to another variable on the stack, the taint will spread to the new variable too. If a register is tainted, the taint follows, ie, it is carried into next instruction unless a non tainted value overwrites the tainted value.

This taint information, as mentioned will be stored in a global list along with a list of tainted registers. This information is then presented to the user at the end of program execution.

4 Evaluation

I first evaluated the tool on the below program. This is a basic program to verify the taint tracking done by the tool.

Program Source :

```
int fun1(char *buf)
{
  char a,b,c;

  a = buf[8];
  b = 10;
  c = buf[0];
  buf[6] = b;
  buf[2] = 9;
  buf[4] = a;
  buf[9] = 0;
  return 0;
}

int main(int ac, char **av)
{
  int fd;
  char *buf;
```

```
  if (!(buf = malloc(32)))
    return -1;

  fd = open("./file.txt", O_RDONLY);
  read(fd, buf, 32), close(fd);
  fun1(buf);
}
```

In the above program, a data is read from a file into a buffer allocated on heap. There is a malloc call to allocate 32 bytes and the read call indicates 32 bytes. Hence, upon read, those 32 bytes belonging to buf should be tainted. Later, few values were explicitly assigned to constants such as the line buf[2] = 9 in fun1. These type of assignments should remove the taints.

The final result obtained confirmed that this part was working correctly.

Program Source :

```
rajiv@rajiv-VirtualBox$ pin -t obj-intel64/new_try.so -- ~/tests/test1
[TAINT] bytes tainted from 0x602010 to 0x602030 (via read)
[READ in 602018]          400619: movzx eax, byte ptr [rax+0x8]
                          eax is now tainted
[WRITE in 7fffffffdc2d]   40061d: mov byte ptr [rbp-0x3], al
                          7fffffffdc2d is now tainted
[READ in 602010]          400628: movzx eax, byte ptr [rax]
                          eax is already tainted
[WRITE in 7fffffffdc2f]   40062b: mov byte ptr [rbp-0x1], al
                          7fffffffdc2f is now tainted
[READ in 7fffffffdc2e]    400636: movzx eax, byte ptr [rbp-0x2]
                          eax is now freed
[WRITE in 602016]         40063a: mov byte ptr [rdx], al
                          602016 is now freed
[WRITE in 602012]         400644: mov byte ptr [rax], 0x9
                          602012 is now freed
[READ in 7fffffffdc2d]    40064f: movzx eax, byte ptr [rbp-0x3]
                          eax is now tainted
[WRITE in 602014]         400653: mov byte ptr [rdx], al
[WRITE in 602019]         40065d: mov byte ptr [rax], 0x0
                          602019 is now freed
[SPREAD]                  400660: mov eax, 0x0
                          output: eax | input: constant
                          eax is now freed
The following registers are tainted at the end of execution

The following memory addresses are tainted at the end of execution
From 602010 to 602011
From 602013 to 602015
From 602017 to 602018
From 60201a to 60202f
From 7fffffffdc2d to 7fffffffdc2d
```

I then tested this tool on a modified version of a simple Pico CTF problem "Overflow1". This tool handles read system calls. Other calls like fgets use read themselves but have an internal buffer. This tool is not designed to handle buffered IO via library functions. Hence I modified the code of overflow1 to test this tool.

Program Snippet :

```
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

void give_shell(){
    gid_t gid = getegid();
    setresgid(gid, gid, gid);
    system("/bin/sh -i");
}

void vuln(){

    char buf[16];
    int fd;
    int secret = 0;
    printf("Read from file\n");

    fd = open("./input.txt", O_RDONLY);
    read(fd, buf, 32);
    close(fd);

    if (secret == 0xc0deface){
        give_shell();
    }
    else{
        printf("The secret is %x\n", secret);
    }
}

int main(int argc, char **argv){

    vuln();
    return 0;
}
```

The above program again is modified to read input from a file. The data is read into buf variable which is 16 bytes long. However, the read system calls allows 32 bytes of data to be read. Hence all 32 bytes of memory would become tainted. This taint region overshoots buf and also taints "secret" which is the goal of the attacker.

The results from testing the tool on the above program indicated that.

Program Snippet :

```
rajiv@rajiv-VirtualBox:$pin -t obj-intel64/new_try.so -- ~/overflow2
Read from file
[TAINT]                 bytes tainted from 0x7fffffffdc10 to 0x7fffffffdc30 (via read)
```

```
The secret is 0
[READ in 7fffffffdc28]  400867: mov rax, qword ptr [rbp-0x8]
                        rax is now tainted
[FOLLOW]                40086b: xor rax, qword ptr fs:[0x28]
[SPREAD]                400896: mov eax, 0x0
                        output: eax | input: constant
                        eax is now freed
[WRITE in 7fffffffdbb0] 7ffff7df04e4: mov qword ptr [rsp], rax
                        7fffffffdbb0 is now tainted
[SPREAD]                7ffff7de9462: mov rax, rdi
                        output: rax | input: rdi
                        rax is now freed
[READ in 7fffffffdbb0]  7ffff7df0536: mov rax, qword ptr [rsp]
                        rax is now tainted
[READ in 7ffff7ffdfc0]  7ffff7ded349: mov rax, qword ptr [rip+0x210c70]
                        rax is now freed
The following registers are tainted at the end of execution

The following memory addresses are tainted at the end of execution
From 7fffffffdc10 to 7fffffffdc2f
```

Similar techniques can also be used to solve more complex buffer-overflow problems and also heap overflow problems. This tool could also be used effectively for ret2reg style attacks since it shows the registers that are tainted.

5 Limitations

As mentioned above, this tool instruments system calls and is not yet designed to handle library calls like fgets, etc. Thus the taint wont be accurate if there is any kind of buffered IO involved.

This tool currently instruments read calls from a file and doesnt take into account other forms of input like environment variables, network, etc.

The main aim of this tool is to provide a memory layout to an attacker so that the attacker would then decide how to exploit a bug. This tool facilitates vulnerability detection but the attacker is still on his/her own while exploiting it.

Also, the information provided by this tool for bigger problems may be overwhelming to the user. Adding a nice visual component that maps the memory is one way to overcome this problem.

6 Related work

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert and David Brumley. "Unleashing MAYHEM on Binary Code"

http://users.ece.cmu.edu/ aavgerin/papers/mayhem-oakland-12.pdf

This paper talks about hybrid symbolic execution which makes use of the best of both online and offline symbolic execution for automatic exploit generation.

David Brumley, Ivan Jager, Edward J. Schwartz, and Spencer Whitman "The BAP Handbook"

http://bap.ece.cmu.edu/doc/bap.pdf

Documentation for BAP which has a detailed description of its capabilities and BAP Intermediate Language

https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

This tool is used for du=ynamic binary instrumentation at assembly level.

http://shell-storm.org/blog/Concolic-execution-taint-analysis-with-valgrind-and-constraints-path-solver-with-z3/

This blog by Jonathan Salwan talks about dynamic taint analysis but using valgrind to instrument the binary.