# CORRUPTING LINUX USERLAND MEMORY FOR FUN AND PROFIT

# WHOAMI

I am AJAY SK

Software Engineer at Larsen and Toubro Infotech Limited.

CTF Player at zh3r0 and TamilCTF

Interested in Low level Security.

# WHAT IS BINARY EXPLOITATION?

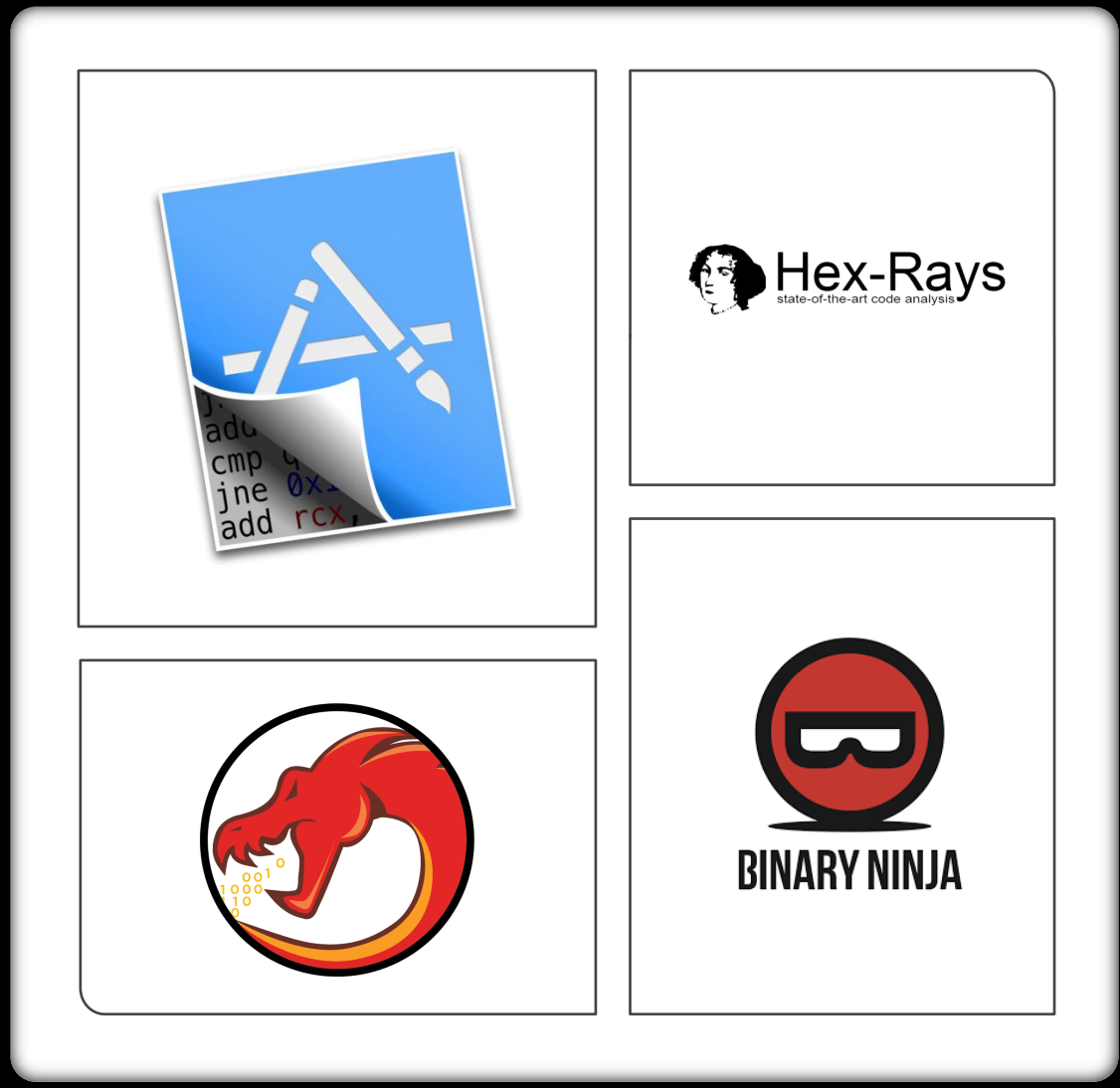
- Binary Exploitation is the process of abusing vulnerabilities that corrupt memory in software we can often rewrite critical application state information in a way that allows us to elevate privileges inside the context of a particular application (like a remote desktop server) or perform arbitrary computation by hijacking control flow and running code of our choosing.

# TOOLS USEFUL FOR BINARY EXPLOITATION

- Disassemblers
- Debuggers
- Decompilers
- Program Tracers
- Fuzzers

# DISASSEMBLERS

- 1)Binary Ninja
- 2)IDA
- 3)Ghidra
- 4)Hopper



# DEBUGGERS

- 1)Cutter
- 2)GDB
- 3)Ollydbg
- 4)Windbg

# DECOMPILERS

- 1)Ghidra's Decompiler
- 2)IDA's Decompiler

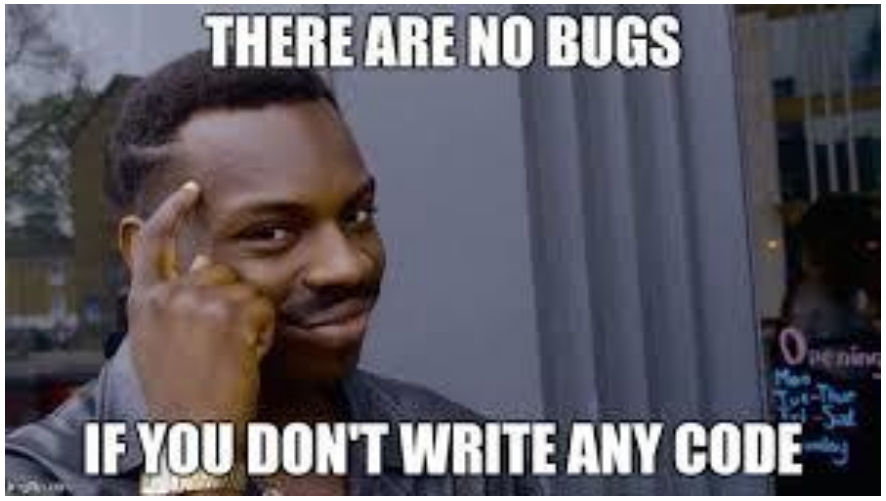
# TRACERS

- 1)Ltrace
- 2)Strace

```
marius@marius-desktop: ~
Usage: ltrace [option ..,] [command [arg ...]]
Trace library calls of a given program.

  -a, --align=COLUMN  align return values in a secific column.
  -A ARRAYLEN         maximum number of array elements to print.
  -c                  count time and calls, and report a summary on exit.
  -C, --demangle      decode low-level symbol names into user-level names.
  -D, --debug=LEVEL   enable debugging (see -Dh or --debug=help).
  -Dh, --debug=help   show help on debugging.
  -e expr             modify which events to trace.
  -f                  trace children (fork() and clone()).
  -F, --config=FILE   load alternate configuration file (may be repeated).
  -h, --help          display this help and exit.
  -i                  print instruction pointer at time of library call.
  -l, --library=FILE  print library calls from this library only.
  -L                  do NOT display library calls.
  -n, --indent=NR     indent output by NR spaces for each call level nesting.
  -o, --output=FILE   write the trace output to that file.
  -p PID              attach to the process with the process ID pid.
  -r                  print relative timestamps.
  -s STRLEN           specify the maximum string size to print.
  -S                  display system calls.
  -t, -tt, -ttt       print absolute timestamps.
  -T                  show the time spent inside each call.
  -u USERNAME         run command with the userid, groupid of username.
  -V, --version       output version information and exit.
  -x NAME             treat the global NAME like a library subroutine.

Report bugs to ltrace-devel@lists.alioth.debian.org
```




# BUGS AND THE VULNERABLE CODE SNIPPET

# BUFFER OVERFLOW

```
char input[16];
puts("Gimme some");
gets(input);
```

No bound check for input size

```
strikerr@pwn-land:/tmp$ ./bof
Gimme some
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: terminated
Aborted (core dumped)
strikerr@pwn-land:/tmp$ ./bof
Gimme some
AAAAAAAAAAAAAAAA
```

Program executes fine when input size is under or equal to the declared size. But Program crashes due to input size larger than declared.

# FORMAT STRING

```
char input[80];
puts("Gimme some");
fgets(input,sizeof(input),stdin);
printf(input);
```

Format Specifier not specified for printf. So the attacker can give his own format specifier to trick the printf for read and write..

```
strikerr@pwn-land:/tmp$ ./fmt
Gimme some
%d %x %p %u %f
2015699044 0 0x563edc5786bf 3218453600 0.000000
strikerr@pwn-land:/tmp$ ./fmt
Gimme some
%p %p %p %p %p %p %p %p %p %p %p %p %p
0x5581534cb6b1 (nil) 0x5581534cb6d8 0x7ffc5ef116e0 0x7c 0x7025207025207025 0x25
20702520702520 0x2070252070252070 0x7025207025207025 0xa20702520702520 0x558152
1ce200 0x7f5f809b82e8 0x5581521ce220
```

Arbitrary memory read

```
strikerr@pwn-land:/tmp$ ./fmt
Gimme some
%n
Segmentation fault (core dumped)
```

Arbitrary memory write

# USE AFTER FREE

```
char *mem1 = malloc(50);
char *mem2 = malloc(50);
memset(mem2,'A',8);
puts(mem2);
if(mem1 != 0 && mem2 != 0)
{
        free(mem1);
        free(mem2);
        puts(mem2);
        fgets(mem2,50,stdin);
        char *mem3 = malloc(50);
        char *mem4 = malloc(50);
```

Reading and Writing on stale pointer.Allowing the attacker to perform arbitrary memory read and arbitrary memory write.

```
strikerr@pwn-land:/tmp$ ./uaf
AAAAAAAA
@r|@QV
BBBBBBBB
Segmentation fault (core dumped)
```

The source code above's last fgets and malloc  leading to arbitrary memory write and last puts leading to heap metadata leak.

# DOUBLE FREE

```
char *mem = malloc(50);
if(mem != 0)
{
        free(mem);
        puts("Double free here!! You may see a error below");
        puts("A double free mitigation kinda here that causes the error");
        puts("But don't be stumbled bcoz of this..");
        puts("There is a way to circumvent this and lead it to RCE");
        free(mem);
```

```
strikerr@pwn-land:~/Corrupting_Linux_UserLand_Memory_for_Fun_and_Profit$ ./double_free
Double free here!! You may see a error below
A double free mitigation kinda here that causes the error
But don't be stumbled bcoz of this..
There is a way to circumvent this and lead it to RCE
free(): double free detected in tcache 2
Aborted (core dumped)
```

You can see here. I freed the memory twice which causes a bug which leads us to perform a arbitrary memory write.

# OUT OF BOUNDS

```
mem = (char *)ptr;
*(_QWORD *)ptr = 0x2A2A2A2A2A2A2A2ALL;
strcpy(mem + 8, "********");
puts(s);
puts("Today, you will have the AMAZING opportunity to edit a string!");
sleep(1u);
printf("But first, a word from our sponsors: %p\n\n", &system);
while ( 1 )
{
  printf(
    "The amazing, cool, epic, astounding, astonishing, stunning, breathtaking, supercalifragilisticexpialidocious string is:\n%s\n",
    (const char *)ptr);
  puts("What character would you like to edit? (enter in 15 to get a fresh pallette)");
  __isoc99_scanf("%ld%*c", &arr_index);
  if ( arr_index == 15 )
  {
    free(ptr);
    ptr = malloc(0x10uLL);
    strcpy((char *)ptr, "****************");
  }
  puts("What character should be in that index?");
  __isoc99_scanf("%c%*c", &value);
  printf("DEBUG: %p\n", (char *)ptr + arr_index);
  *((_BYTE *)ptr + arr_index) = value;
```

The bug here is user provides the index for the array which is not subjected to checks that it is within the size of array.

```
strikerr@pwn-land:~/PWN-Challenges/Imaginaryctf2021/string_edit$ ./string_editor_1
Welcome to StringEditor™!
Today, you will have the AMAZING opportunity to edit a string!
But first, a word from our sponsors: 0x7fdb1a3df290

The amazing, cool, epic, astounding, astonishing, stunning, breathtaking, supercalifragilisticexpialidocious string is:
***************
What character would you like to edit? (enter in 15 to get a fresh pallette)
0
What character should be in that index?
a
DEBUG: 0x5619db8902a0
Done.
The amazing, cool, epic, astounding, astonishing, stunning, breathtaking, supercalifragilisticexpialidocious string is:
a**************
What character would you like to edit? (enter in 15 to get a fresh pallette)
100
What character should be in that index?
b
DEBUG: 0x5619db890304
Done.
The amazing, cool, epic, astounding, astonishing, stunning, breathtaking, supercalifragilisticexpialidocious string is:
a**************
What character would you like to edit? (enter in 15 to get a fresh pallette)
```

Here you can see that the program accepts any index value and places the characters without any checks for the index.

```
    puts("Here ya go! Your string is:");
    puts(target);
    puts("What character would you like to edit? (enter in 15 to see utils)");
    __isoc99_scanf("%ld%*c", index);
    if ( index[0] > 15 )
      break;
    if ( index[0] == 15 )
    {
      puts("1. Admire your string");
      puts("2. Delete your string");
      puts("3. Exit");
      __isoc99_scanf("%ld%*c", index);
      switch ( index[0] )
      {
        case 1LL:
          admire();
          break;
        case 2LL:
          del();
          break;
        case 3LL:
          exit(0);
      }
    }
    else
    {
      puts("What character should be in that index?");
      __isoc99_scanf("%c%*c", &v3);
      target[index[0]] = v3;
      puts("Done.");
    }
  }
  puts("Go away hacker.");
```

Here the index check has been done but kinda incomplete the variable was declared int .They only check for value if it is greater than 15 not checking for lower bound check which is negative values.

```
strikerr@pwn-land:~/PWN-Challenges/Imaginaryctf2021/string_editor_2$ ./string_editor_2
Welcome to StringEditor™!
Today, you will have the AMAZING opportunity to edit a string!
But first, a word from our sponsors: 0x7fffff6c6f6c

Here ya go! Your string is:
***************
What character would you like to edit? (enter in 15 to see utils)
-1
What character should be in that index?
a
Done.
Here ya go! Your string is:
***************
What character would you like to edit? (enter in 15 to see utils)
20
Go away hacker.
```

Here the minus values work but the values greater than 15 won't work. This minus values allows us write values before the location of the variable.

# THE SAD PART OF CODING IN C IS

# BASIC SAFE CODE PRACTISE


PREPARE YOURSELF
SECURE CODING IS COMING

# C SECURE CODING RULES(PART 1)

- Avoid using functions which don't perform buffer checks.
- For functions that uses format strings to represent data. Always specific it,So that it may not lead to format string vulnerabilities.
- Declare number types carefully and if the number is used for any sort of allocation or critical use. Please check for integer overflows and bound check for array index.
- Type confusion or wrong type casting for memory allocation can cause problems.
- Sometimes uninitialized data and referring memory from there can also cause unexpected bugs
- If you are using multithreading for your programs . Please use locking mechanisms for preventing race conditions.

## Heap Rules

### and their bug classes if they get violated

Do not read or write to a pointer returned by malloc[2] after that pointer has been passed back to free.
⇢ Can lead to use after free vulnerabilities.

Do not use or leak uninitialized information in a heap allocation.[1]
⇢ Can lead to information leaks or uninitialized data vulnerabilities.

Do not read or write bytes after the end of an allocation.
⇢ Can lead to heap overflow and read beyond bounds vulnerabilities.

Do not pass a pointer that originated from malloc[2] to free more than once.
⇢ Can lead to double free vulnerabilities.

Do not read or write bytes before the beginning of an allocation.
⇢ Can lead to heap underflow vulnerabilities.

Do not pass a pointer that did not originate from malloc[2] to free.[3]
⇢ Can lead to invalid free vulnerabilities.

Do not use a pointer returned by malloc[2] before checking if the function returned NULL.
⇢ Can lead to null-dereference bugs and occasionally arbitrary write vulnerabilities.

1 Except for calloc, which explicitly initializes the allocation by zeroing it.
2 Or malloc-compatible functions including realloc, calloc, and memalign.
3 free(NULL) is allowed and not an invalid-free, but does nothing.

# C SECURE CODING RULES(PART 2)

# COOL AND OBSCURE ATTACKS

- SROP
- Ret2Dl_Resolve
- Custom Shellcoding
- Heap houses
- Heap feng shui

# The Four Horsemen of Confusion

# Q & A SESSION

# LET'S CONNECT

Itz_me_strikerr#6496

AJAY SK

vital-information-resource-under-siege

Itz_me_strikerr