Hfs-vm

Hfs-vm

In this challenge we were given program emulating custom architecture. After decompilation of binary, we've noticed that it consists of two separate subprograms:

- `kernel` - executing syscalls sent by client
- `client` - loading bytecode of program from `stdin` and emulating it instruction by instruction

They communicates with each other via `socketpair`.

Kernel part

We started analyse of program from the kernel part ('cause it looked simpler). It waits for packet (i.e. syscall command) from client and passes it to the function we've named `process_syscall`. Simplified decompiled code of this function:

```
int process_syscall(byte * packet, ushort * mapped) {
  ushort length;
  int retVal;
  char buffer[18];
  length = * mapped;
  memcpy(buffer, mapped + 1, length);
  switch (*packet) {
  case 0:
    call_ls();
    break;
  case 1:
    call_write(buffer, length);
    break;
  case 2:
    retVal = call_getUIDorEUID(packet[1], buffer, length);
    if (retVal != 0)
```

```
      return -1;
    break;
  case 3:
    retVal = call_readFlag(buffer, length);
    if (retVal != 0)
      return -1;
    break;
  case 4:
    retVal = call_readRandomBytes(packet[1], buffer, length);
    if (retVal != 0)
      return -1;
    break;
  default:
    return -1;
    break;
  }
  memcpy(mapped + 1, buffer, length);
  return 0;
}
```

The most interesting syscall for us was one with `id=3`, which calls function printing flag - `call_readFlag`. As we found our goal in `kernel` part of program, let's move to the `client` part to find a way to use it.

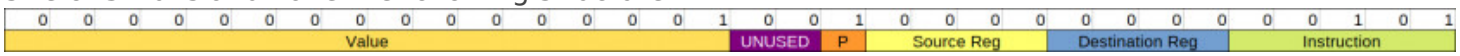
Client part

The current state of program is stored in structure, which we've named `PROGSTAT`:

```
typedef struct PROGSTAT {
  int       fd;
  int       unused;
  void*     mappedArea;
  int16_t   registers[16];
  int16_t   stack[32];
} PROGSTAT;
```

As could be seen, program executed inside VM has 16 registers and 32 bytes of stack. `PROSTAT.fd` is handler to `socketpair` used in communication with `kernel`.

Main part of client program, which we've named `sandbox`, executes each instruction. They all have size of 32-bits and have the following structure:



- `Instruction` - 5 bit number defining type of instruction
- `Destination` - destination register
- `Source Reg` - source register
- `P` - parameter, specyfing if source argument was `Source Reg` of `value`

Client implements 10 instructions:

- mov
- add
- sub
- swap
- xor
- push
- pop
- change_element_on_stack
- get_element_from_stack
- syscall
- dump

For us the most interesting were: `mov`, `push`, `syscall` and `dump`. The selection of syscall is made by specifing its value in the first register - `r1`. We've used `mov` command to achive this: `Value=3`, `Destination=1`, `P=1` (use value as source instead of register). Before calling `syscall`, we need to decrease `SP` (stack pointer) so `syscall` command has a place to write the flag. We've achived this by calling 25 times `push`. After this we've called `syscall` (it doesn't requires any arguments - just a valid instruction code) and `Kernel` wrote flag on the stack. To print it we've used `dump` command (again, no arguments requried), which prints on `stdout` values of all registers and dumps all 32-bytes of stack.

Final "payload":

```
from pwn import *
push =      p32(0b00000000000000001001000000000101)payload =
p32(0b00000000000000011001000000100000) + \ # Mov r1, 3
push*0x19                               + \ # Push * 25
p32(0b00000000000000000000000000001001) + \ # Syscall
p32(0b00000000000000000000000000001010)     # Dump
print payload
```

Revision #8

Created Mon, Apr 8, 2019 5:13 PM by 3na10

Updated Mon, Dec 16, 2019 7:17 PM by Lorak_