Prologue

This is a challenge in the zer0pts CTF 2022. It's pity that I didn't solve it in the game, but it's supper worthy to write a singel post for this challenge. I learned a lot in this challenge, including some unrelated knowledge. Yeah, I tried tons of wrong ways to solve it. And the official solution is really innovative for me, I guessed that but to be honest, I can't confirm it because I never met this type of exploiting. It's a long story, let's start the awesome trip!

Analysis

main

zer0pts gives the full set-up enviroment, including the source code. I think this type of pure pwn is better for some challenges cuz people could reproduce the challenge easily.

```
int main() {
  ...
  if (cpid == 0) {

    /* Child process: sandboxed */
    close(c2p[0]);
    close(p2c[1]);
    setup_sandbox();
    child_note(c2p[1], p2c[0], ppid);

  } else {

    /* Parent process: unsandboxed */
    close(c2p[1]);
    close(p2c[0]);
    parent_note(c2p[0], p2c[1], cpid);
    wait(NULL);

  }
}
```

In the main function, the program would creat two process, child and parent. And use the pipe to implement the communication between the child and parent process. Also, the child process is protected by bpf sandbox.

sandbox

I used the `seccomp-tools` to check the sandbox rules. And find there is a black list of systcalls. Except this flaw, this bpf is good in my view because it bans the i386 syscalls and large syscalls(+0x40000000).

```
$bin seccomp-tools dump ./chall
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x00 0x11 0xc000003e  if (A != ARCH_X86_64) goto 0019
 0002: 0x20 0x00 0x00 0x00000000  A = sys_number
 0003: 0x35 0x0f 0x00 0x40000000  if (A >= 0x40000000) goto 0019
 0004: 0x15 0x0e 0x00 0x00000002  if (A == open) goto 0019
 0005: 0x15 0x0d 0x00 0x00000101  if (A == openat) goto 0019
 0006: 0x15 0x0c 0x00 0x0000003b  if (A == execve) goto 0019
 0007: 0x15 0x0b 0x00 0x00000142  if (A == execveat) goto 0019
 0008: 0x15 0x0a 0x00 0x00000055  if (A == creat) goto 0019
 0009: 0x15 0x09 0x00 0x00000039  if (A == fork) goto 0019
 0010: 0x15 0x08 0x00 0x0000003a  if (A == vfork) goto 0019
 0011: 0x15 0x07 0x00 0x00000038  if (A == clone) goto 0019
 0012: 0x15 0x06 0x00 0x00000065  if (A == ptrace) goto 0019
 0013: 0x15 0x05 0x00 0x0000003e  if (A == kill) goto 0019
 0014: 0x15 0x04 0x00 0x000000c8  if (A == tkill) goto 0019
 0015: 0x15 0x03 0x00 0x000000ea  if (A == tgkill) goto 0019
 0016: 0x15 0x02 0x00 0x00000136  if (A == process_vm_readv) goto 0019
 0017: 0x15 0x01 0x00 0x00000137  if (A == process_vm_writev) goto 0019
 0018: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0019: 0x06 0x00 0x00 0x00000000  return KILL
```

One key of this challenge is the backlist filter, I guessed that the author may want to share with us some attacking skills with some syscalls. But I faild to find that syscall during the game. In the game, I thought, that syscall may be related to the process control so that I go through all the syscalls who have pid related parameters and all the syscalls related to the "open".

Interface and Core

```
void child_note(int c2p, int p2c, int ppid) {
  int res;
  request_t req;
  uint64_t value;

  print("1. new\n");
  print("2. set\n");
  print("3. get\n");
  ...
}
```

In the sandbox, the child process would provide an interface of the parent process. There is a manual in the child proces. The child would take the input from user. According to different inputs, the child process would generate a call different functions on the parent process. Btw, there is a buffer overflow in the child process but we can't use it to get the flag because of the sandbox.

For this type of challenge, we usually try to control the child and make it to be a bad boy so that we can use it to send some evil api calls to trigger the vulnerability in the parent process.

```
while (1) {
  if (read(c2p, &req, sizeof(req)) != sizeof(req))
    return;

  switch (req.cmd)
  {
    /* Create new buffer */
    case NEW: {
  ...
```

For the parent process, it would take the message from its child and parse it. But in my view, the parse part is secure and there are so many checks to prevent bad value. There is no trust between this father and son.

```
      ...
      old = buffer;
      if (!(buffer = (uint64_t*)malloc(req.size * sizeof(uint64_t)))) {
        /* Memory error */
        size = -1;
        RESPONSE(-1);
        break;
      }
      ...
```

But some thing strange catch my attention after reading the code for 3 hours. In the `new` feature, the size could be set to -1 and it's an `unsigned int` which means we could write/read arbitrary address by `SET/GET`. However, the story is not such easy.

I spent another 3 hours to review the malloc's source code to find a way to return the "NULL". In my memory, malloc would return a negtive number on fail and return the address of chunk when successing. And this 3 hours proves that, we have no way to return a 0 by construct a bad api call.

Solution

1. Use Buffer overflow to control the child process
2. Leak the libc base. we can use it in the parent process because the child process would inherit the memory layout of the parent.
3. Use some syscall in child to make parent's malloc return 0
4. Send some api call to write/read arbitrary address to hijack the parent process

We can combine two parts, the blacklist-filter and the `size=-1` to generate above solution but I can't implement during the game because I can't find a syscall to limit the parent process and I find the keystone of my solution in the official exp. That's an syscall I missed in the game, prlimit64.

limite a process

seccomp/BPF