TRAPS

CSCG 2023 QUALIFIERS

A flag checker. Cheap entertainment.

1 INTRODUCTION

traps is a simple flag checker that accepts the flag on standard input and verifies whether it is correct. The primary trick in *traps* is that it extensively uses `ptrace` to dynamically load shellcode and foil debugging attempts. This tends to be quite effective, since essentially all Linux debuggers also use `ptrace`, and there can only ever be one tracer for a target prcoess[1].

2 SOLUTION

Initially, the binary forks a child process that attaches to the parent, sets `r15` to zero, and then quits. This overwrites the result of `wait` (which should never be zero) and bypasses a red herring "flag check" that doesn't actually check the flag.

```
void __fastcall step_1(unsigned int pid)
{
    int status; // [rsp+Ch] [rbp-FCh] BYREF
    struct user_regs_struct regs; // [rsp+10h] [rbp-F8h] BYREF
    if (!(unsigned int)fork()) {
        if ( ptrace(PTRACE_ATTACH, pid, 0LL, 0LL) >= 0 ) {
            if ((waitpid(pid, &status, __WALL) & 0x80000000) != 0LL
                    || !__OFSUB__((status & 0x7F) + 1, 1) && (status & 0x7F) != 0
                    || (status & 0x7F) == 0 )
                exit(BYTE1(status));
            if (ptrace(PTRACE_GETREGS, pid, 0LL, &regs) >= 0) {
                regs.r15 = 0LL;
                if (ptrace(PTRACE_SETREGS, pid, 0LL, &regs) >= 0
                        && ptrace(PTRACE_DETACH, pid, 0LL, 0LL) >= 0)
                    _Exit(0);
            }
        }
        exit(-1);
    }
    if (wait(0LL)) { // This is actually the r15 written via ptrace
        puts("Welcome to your average flag checker! Please give me a flag:");
        fgets(&regs, 64, stdin);
        red_herring_check(&regs);
    }
}
```

The actually interesting part comes next. The binary spawns another child process. The parent waits to be debugged (via `PTRACE_TRACEME`), creates a page that is both writable and executable via `mmap`, copies some shellcode to it, and jumps there.

Meanwhile, the child process starts debugging: After setting `PTRACE_O_EXITKILL`, it starts a somewhat intricate dance with the parent-debuggee: Whenever the debuggee suspends (`waitpid` returns), it updates the `rdi` and `rsi` registers (from an array I call `arguments`), and copies some data (from `poke_data`, with the amount depending on the `sizes` array) into the debuggee. Since this writes at `rip + 0x86`, it stands to reason that this modifies the shellcode that was started in the parent earlier.

```
    int index = 0;
    size_t poke_offset = 0;
    while (sizes[index]) {
        waited_pid = waitpid(pid, &status, __WALL);
        if (CHECK(waited_pid, status) /* Details don't matter */)
```

[1]A short plug for `plutonium-dbg` seems in order at this point, even though I didn't use it for this challenge.

```
            goto exit_child_status;
        if (ptrace(PTRACE_GETREGS, pid, 0LL, &regs) < 0)
            exit(-1);

        // Load 16 bytes from the arguments data into rsi and rdi.
        * (__m128 *) &regs.rsi = _mm_loadh_ps(&arguments[index]);
        if (ptrace(PTRACE_SETREGS, pid, 0LL, (__int64)&regs) < 0)
            exit(-1);

        // Write to memory
        int bytes_to_poke = sizes[index];
        char *poke_at = regs.rip + 0x86;
        for (int offset = 0; offset < bytes_to_poke; offset += 8)
            if (ptrace(PTRACE_POKETEXT, pid, poke_at + offset, &poke_data[poke_offset +
              offset]) < 0)
                exit(-1);

        poke_offset += bytes_to_poke;
        ++index;
        if (ptrace(PTRACE_CONT, pid, 0LL, SIGCONT) < 0)
            exit(-1);
    }
```

Unfortunately, the data in the binary appears encrypted, so let's take a look at the initial shellcode which is still "readable" (after some massaging — certainly IDA didn't like it very much):

```
__int64 __fastcall shellcode(unsigned __int64 a1, __int64 a2)
{
    void *S;
    __int64 i;
    BYTE j;

    __debugbreak();
    S = alloca(256LL);
    i = 256LL;
    do {
        S[i - 1] = i;
    } while (--i);
    j = 0LL;
    for (i = 255LL; i >= 0; --i) {
      j = ((a1 >> (8 * ((unsigned __int8)i & 7u))) + S[i] + j);
      swap(S[j], S[i]);
    }

    j = 0;
    do {
      ++i;
      j = S[i] + j;
      swap(S[i], S[j]);
      ((_BYTE *)__next)[i - 1] ^= S[S[i] + S[j]];
    } while ( a2 != i );
    return __next();
}
```

Clearly, this is RC4, with the key in `a1` and the length of the data in `a2` — except that's not quite right: The initial setup of the permutation is different from what it should be. Instead of having the values 0 through 255, it is rotated left by one entry, so it contains 1 through 255 and then a single 0.

When the code hits the `__debugbreak()` (IDA-speak for an `int3` software breakpoint instruction), the tracer updates key and size, and starts copying data to where `__next()` is.

For initial analysis, to grab the actual values that go through `ptrace`, I built a small `strace` equivalent that does not rely on `ptrace`. Instead, it uses the `seccomp_unotify` mechanism to intercept system calls before they are executed[2].

[2] I will publish code for this at `https://gitlab.com/tobiasholl/minitools` at some point after the end of CSCG, stay tuned.

When the debugger performs a PTRACE_SETREGS operation, I pull the new register set from its /proc/<pid>/mem and store rdi (the secret key) and rsi (the length of the data) for later. Since there are many calls using PTRACE_POKEDATA, I found it easier to instead just grab the full ciphertext (plus the decryption shellcode) from the rwx mapping in the debuggee whenever a PTRACE_CONT is performed.

Then, we can easily decrypt each stage of the shellcode by simply re-implementing this modified RC4 in Python.

1. The first shellcode stage simply writes the flag checker prompt and reads the flag from standard input:

```
sys_write(1, "Welcome to your average flag checker! Please give me a flag:\n", 0x3e)
rax = bytes_read = sys_read(0, flag = alloca(0x40), 0x40)
push(rax)
```

2. Then, it checks the length of the flag and XORs every byte with 13:

```
if bytes_read != 0x40:
    sys_write(1, "NO :(\n", 7)
    sys_exit(1)
for rcx in range(0x40, 0, -1):
    flag[rcx - 1] ^= 0xd
```

3. The third stage sets up additional code after the shellcode that is typically decrypted, at offset 255:

```
r15 = find(__return_address(), rbx = 0xd39fc7066b8bda5c) + 8 # == &operations[0]
r14 = rax = sum(operations[:4]) # == 0x20c
r15 -= 8 * (1 + strlen(operations[4:])) # == 0x90 [=> r15 == &arguments[4]]
r13 = &operations[4]
memcpy(&shellcode[0xff], &shellcode[0x70], 0x61)
```

The newly-added code does some peculiar things. In particular, it modifies the decryption routine to remove the breakpoint that triggers waitpid in the tracer, and changes the key stream generation.

```
decryption[0] = 0x90 # int3 => nop
decryption[0x4f] = 0x90 # mov al, 0 => mov al, 0x90 (initial j in key stream generation)

if operations[index] == 0: # via r13
    sys_exit(0)

rsi = endof(&operations[index]) + 1 + r14 # => rsi == &poke_data[poke_offset]
memcpy(&shellcode[0], rsi, operations[index])

r14 += operations[4] # => r14 == poke_offset
decrypt_size = operations[index]
decrypt_key = arguments[index]
r13 += 1 # => &operations[index + 1]
```

From stage 4, instead of jumping to the breakpoint at the start of the decryption routine to set up the next stage through ptrace, all later stages will instead jump to this code. This will load the next stage immediately, without the round-trip through ptrace.

4. Before we finally get to the actual point behind this challenge, there is yet another anti-debugging check. The debuggee now attempts to attach to its own debugger (which is usually allowed, unless the debugger is being debugged). This probably serves to hide the fact that no further calls to ptrace will take place.

```
if ptrace(PTRACE_SEIZE, tracer_pid, 0, 0) != 0: # tracer_pid is in r12
    sys_write("NO :(\n", 7)
    sys_exit(1)
```

Stages 5 through 20 are very similar. They all perform the same actions with different values:

```
coeffs = (uint32_t *) &shellcode[0x3b] # Different values in each stage
constant = ... # Different values in each stage
edi = (sum(flag.u32[i] * coeffs[i] for i in range(0x10)) == constant) ? 1 : 0
push(pop() + edi)
```

The reverse engineering afficionado already recognizes this as matrix-vector multiplication. Each stage computes the dot product between the coefficients (a row in the matrix) and the flag vector. If the result matches the constant value, we increment the value at the top of the stack (remember, it started with the number of bytes read in stage 1, and from stage 2 we know that that is 0x40).

Before we pull the matrix out of the binary, let's briefly take a look at the final stage. As expected, it checks that all 16 entries of the multiplication result are correct.

```
str = (pop() == 0x50) ? "YES !\n" : "NO :(\n"
sys_write(1, str, 6);
sys_exit(0)
```

Then, we can simply use `sagemath` to solve the matrix equation in $\mathbb{Z}/2^{32}\mathbb{Z}$

```
#!/usr/bin/env sage
import struct
Zq = Zmod(2^32)

MM = [
  # ... Coefficients here, see section 3 for the full code
]
VV = [
  # ... Constant values here, see section 3 for the full code
]

solution = matrix(Zq, MM) \ vector(Zq, VV)
raw_bytes = struct.pack('16I', *list(solution))

# Remember the XOR from the start
print(bytes(b ^^ 0xd for b in raw_bytes).decode())
```

and get the flag: `CSCG{4ND_4LL_0FF_TH1S_W0RK_JU5T_T0_G3T_TH1S_STUUUP1D_FL44G??!!1}`

3 CODE

The hook (`hook.sh`) used with `ministrace` also uses `minicore` (which I built for the *Ghost Flag* challenge) to dump the entire memory of the target, but `dd` would have been enough.

```
#!/bin/bash
set -euo pipefail
src="$(dirname "${BASH_SOURCE[0]}")"
tools="~/Documents/code/minitools/minicore"

if [ "$3" -ne 101 ]; then
    exit 0 # not ptrace
fi

if [ "$4" != "0x7" ] && [ "$4" != "0xd" ]; then
    exit 0 # not PTRACE_CONT or PTRACE_SETREGS
fi

pid="$(($5))"
if ! grep -q 'rwxp[^s]*$' /proc/"$pid"/maps; then
    exit 0 # no rwx mapping
fi

coref="$(mktemp)"
function finish {
    rm -f -- "$coref"
}
trap finish EXIT

if [ "$4" == "0x7" ]; then
    # PTRACE_CONT
    addr="$(grep 'rwxp[^s]*$' /proc/"$pid"/maps | sed 's/-.*//')"
```

```
    "$tools/minicore" --no-suspend "$pid" "$coref" 2>/dev/null # suspended by virtue of ptrace
    python "$tools/minicore-parse.py" --only "0x$addr" "$coref" | python "$src/decrypt-current.py"
elif [ "$4" == "0xd" ]; then
    # PTRACE_SETREGS
    tracerpid="$1"
    regaddr="$7"
    fromaddr="$(($regaddr+0x68))"
    toaddr="$(($regaddr+0x78))"
    "$tools/minicore" --no-suspend "$tracerpid" "$coref" 2>/dev/null # suspended in ministrace
    python "$tools/minicore-parse.py" --range "$fromaddr" "$toaddr" "$coref" > "$src/rsirdi.bin"
fi
exit 0
```

Decryption (`decrypt-current.py`) is just collecting the latest `rdi` and `rsi` and reimplementing the modified RC4 from the binary.

```python
#!/usr/bin/python3
import os
import struct
import sys

initial = int(sys.argv[1], 0) if len(sys.argv) > 1 else 0 # usually normal
skip = int(sys.argv[2], 0) if len(sys.argv) > 2 else 0x87 # usually with preamble
name = sys.argv[3] if len(sys.argv) > 3 else None

encrypted = bytearray(sys.stdin.buffer.read()[skip:])

os.chdir(os.path.dirname(os.path.abspath(__file__)))
with open('rsirdi.bin', 'rb') as rsirdi:
    raw = rsirdi.read()[:0x10]
    rsi_b, rdi_b = raw[:0x8], raw[0x8:]
    rsi_i, rdi_i = struct.unpack('QQ', raw)
os.unlink('rsirdi.bin')

S = [e % 256 for e in range(1, 257)] # lol
j = 0
for i in range(255, -1, -1):
    j = (rdi_b[(i+1) & 7] + S[i] + j) % 256
    S[i], S[j] = S[j], S[i]

j = initial
for ii in range(rsi_i):
    i = ii + 1
    assert i < 256
    j = (S[i] + j) % 256
    S[i], S[j] = S[j], S[i]
    encrypted[ii] ^= S[(S[i] + S[j]) % 256]

existing = [e for e in os.listdir('.') if e.startswith('shellcode')]
of = f'shellcode-{len(existing):02d}' if name is None else name
with open(of, 'wb') as dec:
    dec.write(encrypted[:rsi_i])

os.chown(of, 1000, 1000)
```

To grab the later stages that don't come through `ptrace`, we can simply fetch the data from the binary. After all, at this point we know how the decryption works.

```python
#!/usr/bin/python3
import struct
import tempfile
import os
import sys

index = int(sys.argv[1]) # starts to be sane at 4

with open('traps/traps', 'rb') as elf:
```

```
    binary = elf.read()

arguments = binary[0x1915:0x19c5]
operations = binary[0x19c5:0x19db]

data_off = 0x19db
data_off += sum(operations[:index])

data = binary[data_off : data_off + operations[index]]

rdi = arguments[index * 8: index * 8 + 8]
rsi = operations[index]

with open('rsirdi.bin', 'wb') as rr:
    rsi = struct.pack('Q', rsi)
    rr.write(rsi + rdi)

with tempfile.NamedTemporaryFile() as tf:
    # yikes
    tf.write(data)
    tf.flush()
    os.system(f'cat {tf.name} | python decrypt-current.py 0x90 0 shellcode-{index:02d}')
```

Extracting the values from the decrypted shellcode is simple:

```
#!/usr/bin/python3
import sys
import struct

with open(sys.argv[1], 'rb') as shellcode:
    args = struct.unpack('17I', shellcode.read()[0x3b:0x3b+0x40+4])

print('[' + ', '.join(hex(value) for value in args[:0x10]) + ']')
print(hex(args[-1]))
```

The final sagemath solver:

```
#!/usr/bin/env sage
import struct
Zq = Zmod(2^32)

MM = [
    [0xa328007, 0xc94147c0, 0xb76455ca, 0xe0d68f61, 0x4495b889, 0x26374927, 0xc286c90,
     0x220ce3c6, 0x24bd72e2, 0x1df2b113, 0x40e0ebe, 0xf3896734, 0x2198221c, 0x363e41a5,
     0xaf3718bd, 0x9a665bd0],
    [0x4dec6150, 0x90cb1283, 0xb83a1e93, 0x498d1ad1, 0x177e2e4a, 0x5ef4a184, 0x659cc092,
     0xa1259489, 0xb29770a7, 0x8ca2197a, 0xc2e14012, 0x5f439cc3, 0x5b8ef854, 0x47b4c616,
     0x91c69756, 0x8d93610a],
    [0x4e4f744f, 0x72f06215, 0xa158e802, 0xe8e3dd2, 0xda501bb9, 0x35e109a2, 0x497172f0,
     0xc2b05062, 0xb5ac1652, 0xef609a18, 0x9dead27b, 0x8825b544, 0xafd38f9b, 0x8be84dc1,
     0xada4520, 0x141a6e1d],
    [0x5b471250, 0x81637dc3, 0xbb790c62, 0xd8c88f22, 0xe3280002, 0x27d89aa9, 0xc8ede042,
     0x359e6ff3, 0xf01ea2ae, 0x3bb858f5, 0x6a2d647f, 0x3c51a9a, 0x1816cf55, 0xd919183b,
     0xe0750936, 0xc9d93460],
    [0xef6cc5c8, 0x2d153dd2, 0xbdeac3b, 0x2ddb2de3, 0x74ce3d45, 0x10a9fbfe, 0x1bb7483,
     0xc9103345, 0x3fa915e, 0xe2974833, 0x38919ba, 0x713d13f9, 0x783fd5ce, 0x7d265458,
     0x44668d2e, 0xed23c2b0],
    [0x5a0a916c, 0x756fedb2, 0xf2bc35d4, 0xf7797e3e, 0xc281de27, 0xd3714768, 0x75f0a2a9,
     0xcf6548e0, 0xbb61b030, 0x8870a03b, 0x429ea9ab, 0x1dca5878, 0xb933a576, 0x82b4fe93,
     0x8a492ebc, 0x44cc7c45],
    [0xc8724b91, 0xac1c5a40, 0x7b2495f7, 0x40e71e4a, 0x538c9188, 0xc31158b8, 0xce92d376,
     0xc155f3a7, 0x5bd5c8ef, 0x7b68cece, 0xf8ac9ecc, 0xeed46c40, 0xf2275fd1, 0x6a46bc26,
     0xca74d62e, 0x9d426211],
    [0x75cde7bc, 0x1cc65ba6, 0xe7e365e8, 0x7cf5a15f, 0xf801f05e, 0x1c6d0bd1, 0xdba5003c,
     0xf5c015f7, 0xb6eca7f3, 0x8bdcd9db, 0x90b67e66, 0x8530cf38, 0xd114145f, 0x5b218747,
     0x1c8e277f, 0x29911567],
    [0x603c406d, 0x1243b64d, 0xce550d28, 0x27be86c7, 0x63cabf2, 0xab1aab5e, 0x28e718ad,
```

```
     0x43c45e1c, 0x954e402e, 0x9ca8fcd2, 0xade23477, 0xea5d9d6b, 0x2e7270ed, 0x8ef52390,
     0xd0f0447f, 0x205f8a40],
    [0x5ab7ca7f, 0x52ae61b3, 0x1fe1b78a, 0x82b557ad, 0x2dcadeac, 0xa25c6a68, 0x38d94f9e,
     0x4abcb6db, 0xad126d8c, 0xd0f91341, 0xdbf1cce, 0x652a7cb6, 0x56e2ceab, 0xed341908,
     0xe0554232, 0xae09176b],
    [0x45aeb379, 0x75dd2502, 0x64746741, 0x850b7746, 0x83e80d75, 0x9cf2433d, 0x97e5fa04,
     0x8374d300, 0xafb800dc, 0xaa839145, 0x5be1f027, 0x260af107, 0xbcb61b83, 0x535451c0,
     0xb27f1556, 0x31dd1e8e],
    [0x246164ee, 0xb3deb0aa, 0x3ef552cc, 0x32dbd2c7, 0x9fcf0928, 0x4e6e396d, 0x21670a7e,
     0xee7d8e8a, 0x458330fa, 0x7f0f1925, 0xd5c08e54, 0x822ea831, 0x5ea461c0, 0x61c400dd,
     0xf5e1968b, 0x36d8e19d],
    [0x9cd7f1c, 0xabd03d78, 0x75aca063, 0xd76dcf09, 0x47fa6b34, 0xb90912cd, 0x99598aca,
     0x2fef8e94, 0x57ec3e2, 0x7173e6d9, 0xa3e1877a, 0x14b2d8a6, 0xfcc3501c, 0x6287de81,
     0x498ed1f7, 0x2b831194],
    [0xee1d7d34, 0xd7677ea6, 0xc0a83ae6, 0x53bee433, 0xf6997363, 0x903e9f27, 0x4321f9f1,
     0xbc868c7, 0x53e9febc, 0x2d751d64, 0x86ff4195, 0x8d5447db, 0x95a91ba1, 0xa74dfca0,
     0x833eb53e, 0x55f65946],
    [0x83638120, 0xc7d7632e, 0xf4991244, 0xc5e95e40, 0x3119b6c, 0x7cd3831c, 0x3abad4d4,
     0x4efb44cf, 0xacbd9e23, 0x7ba02375, 0x366dcc15, 0xc884671e, 0x64f990cb, 0xc542deff,
     0xe0fec07, 0xf27dbfcb],
    [0xf68e23b2, 0xe81e9cd6, 0x63a77c02, 0x62a43a71, 0xcadebfb8, 0xd27f4a2e, 0x90a13e3,
     0x31bd71af, 0x7ebeec3f, 0x22363530, 0xd918c728, 0x90c71ba0, 0xfba8857e, 0x839687e8,
     0xff4ac110, 0xd6a289ee],
]
VV = [
    0x9b1bf3e5, 0x8e504d15, 0x57ae4bcd, 0xb9dcdb81, 0x50f37921, 0xda39079c, 0x7b650f77,
    0x8968c084, 0x3ab87a33, 0x829801a6, 0x377f1cc1, 0xccae639, 0x4b91198b, 0x7d433d2,
    0x9d63b107, 0xeb921216
]

solution = matrix(Zq, MM) \ vector(Zq, VV)
raw_bytes = struct.pack('16I', *list(solution))

# Remember the XOR from the start
print(bytes(b ^^ 0xd for b in raw_bytes).decode())
```