

KFuzzTest

One Year On



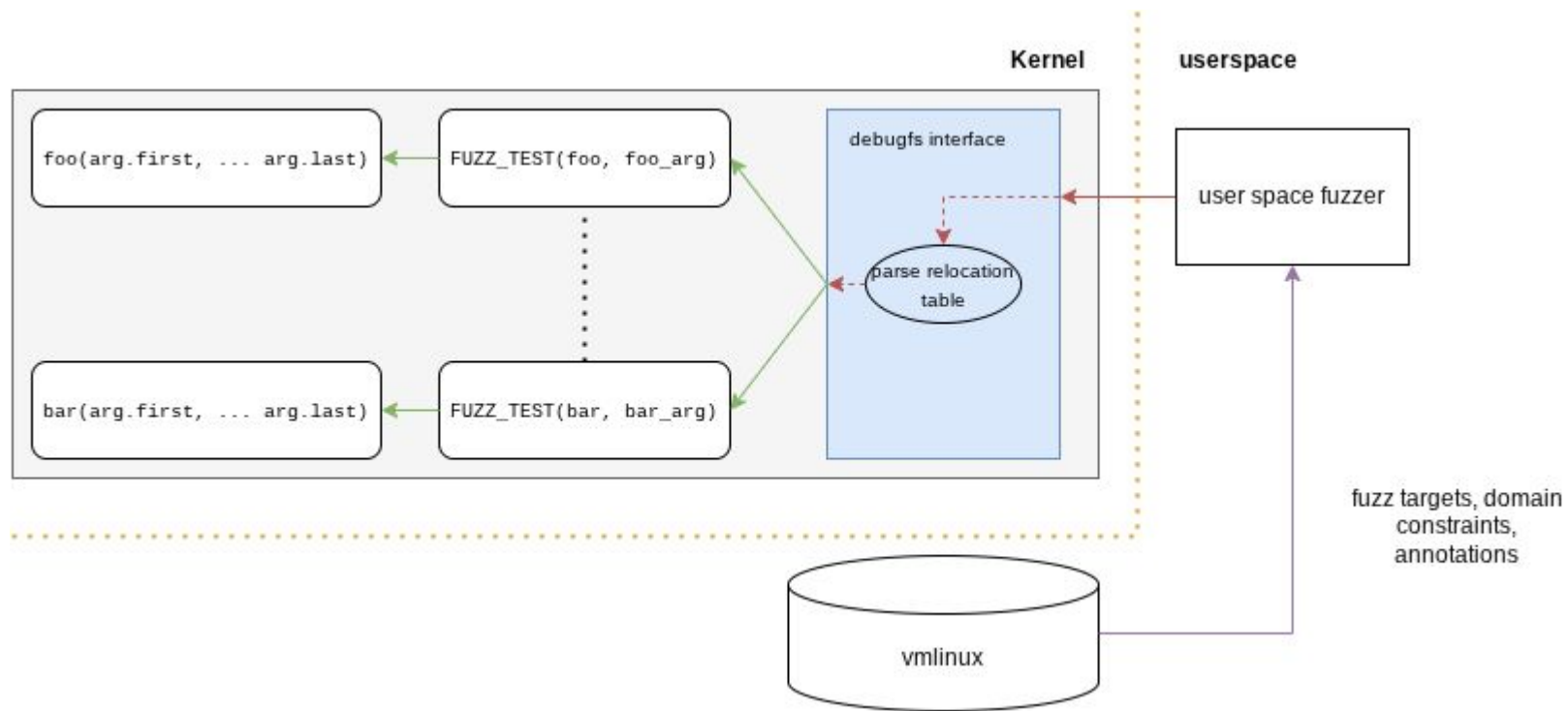
Ethan Graham *ETH Zürich*

Recap

	Test Case Discovery	Targeting deep code directly
Syscall Fuzzing	Yes	Coarse by design
Unit Testing	No, doesn't try to	Yes

Can we get both? → KFuzzTest

Design



The API, circa Late 2025

```
struct pkcs7_message *pkcs7_parse_message(const void *data, size_t datalen);
```

```
struct pkcs7_parse_message_arg {  
    const void *data;  
    size_t datalen;  
};
```

```
FUZZ_TEST(test_pkcs7_parse_message, struct pkcs7_parse_message_arg)  
{  
    KFUZZTEST_EXPECT_NOT_NULL(pkcs7_parse_message_arg, data);  
    KFUZZTEST_ANNOTATE_ARRAY(pkcs7_parse_message_arg, data);  
    KFUZZTEST_ANNOTATE_LEN(pkcs7_parse_message_arg, datalen, data);  
  
    pkcs7_parse_message(arg→data, arg→datalen);  
}
```

Source: [Patch v3](#)

The API, circa Early 2026

```
FUZZ_TEST_SIMPLE(test_pkcs7_parse_message)
{
    pkcs7_parse_message(data, datalen);
}
```

Source: [Patch v4](#)

Areas for improvement

- Input serialization
 - Kernel side complex deserialization
 - Fuzzer side complex serialization logic
- Does KFuzzTest need to be standalone?
- Can we find real bugs with stateless BLOB fuzzing?

Directions to explore

1. Stateful fuzzing
2. Fuzzing harnesses in KUnit

Directions to explore

1. **Stateful fuzzing**
2. Fuzzing harnesses in KUnit

Experiment

Agents are good at navigating source files and writing fuzz harnesses.

1. Let an agent search for stateless APIs that take BLOBs as arg
2. Let it run a fuzzing campaign on the harnesses
3. Report any results

Result: a few false positives, but **no real bugs**.

Stateful Fuzzing Prototype

Idea: let the fuzzer call a sequence of operations on persistent state.

- Initialize `st`
- `n` times:
 - Run `op(st, arg)`
 - Check `is_correct(st)`
- Teardown

→ *Logic bugs can be detected on top of sanitizers and crashes.*

First API Proposal

```
KFUZZ_REGISTER_HARNESS(ida, kfuzz_ida_initialize, kfuzz_ida_teardown,  
    kfuzz_ida_check_correctness,  
    kfuzz_ida_op_alloc_range,      /* 0 */  
    kfuzz_ida_op_alloc_min,       /* 1 */  
    kfuzz_ida_op_free,            /* 2 */  
    kfuzz_ida_op_find_first,      /* 3 */  
    kfuzz_ida_op_check,           /* 4 */  
    kfuzz_ida_op_destroy)         /* 5 */
```

Each op ~= old KFuzzTest harnesses, but they all operate on the same shared state

Found a bug!

```
[ 20.956513] kfuzz_ida: op 24: IDA enumerated 4 allocated IDs, the shadow holds 5
[ 20.957778] WARNING: lib/kfuzz_ida.c:334 at kfuzz_ida_op_check+0x2ee/0x5f0, CPU#1: dd/413
[ 20.959160] Modules linked in:
[ 20.959761] CPU: 1 UID: 0 PID: 413 Comm: dd Tainted: G                N 7.0.0-rc5-g101d81c7b83b #133 PREEMPT(full)
[ 20.961584] Tainted: [N]=TEST
[ 20.962124] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux 1.17.0-2-2 04/01/2014
[ 20.963926] RIP: 0010:kfuzz_ida_op_check+0x2fa/0x5f0
```

Which was already be identified, and fixed by [this patch](#)

Directions to explore

1. Stateful fuzzing
2. **Fuzzing harnesses in KUnit**

Prototype Design

- Implement harnesses as a subtype of KUnit tests
- Use debugfs to get inputs from userspace
- Inputs run on fresh `kunit_try_catch_run` threads

KUnit Fuzzing Harnesses API

```
#include <crypto/pkcs7.h>
#include <kunit/test.h>

static void fuzz_pkcs7_parse_message(struct kunit *test, char *data,
                                     size_t datalen)
{
    pkcs7_parse_message(data, datalen);
}

static struct kunit_case pkcs7_test_cases[] = {
    KUNIT_CASE_FUZZ(fuzz_pkcs7_parse_message),
    {}
};

static struct kunit_suite pkcs7_test_suite = {
    .name = "pkcs7",
    .test_cases = pkcs7_test_cases,
};

kunit_test_suites(&pkcs7_test_suite);
```

Summary: make fuzzing as easy as unit testing

- Is anyone using fuzzing to target specific code?
- Does stateful fuzzing have the right shape?
- KUnit:
 - Any benefit in letting userspace pass in test inputs?
 - Throughput: how should threading be handled?
 - Should we bring back serialization for generality?



Backup Slides

ida reproducer

```
DEFINE_IDA(ida);
```

```
/* 1. One live ID in chunk 0. This matters only because an IDA chunk's  
 * XArray entry exists if and only if the chunk holds >= 1 live ID  
 * (ida_free() stores NULL once the bitmap empties). So this is what  
 * makes chunk 0 "present".  
 */
```

```
ida_alloc_range(&ida, 0, 0, GFP_KERNEL);          /* == 0    */
```

```
/* 2. One live ID in chunk 1. IDA_BITMAP_BITS == 1024 on 64-bit. */
```

```
ida_alloc_range(&ida, 1024, 1024, GFP_KERNEL); /* == 1024 */
```

```
/* 3. The lowest used ID in [1, 2047] is 1024. */
```

```
id = ida_find_first_range(&ida, 1, 2047);
```

```
/* expected: 1024
```

```
 * actual:   -ENOENT */
```

Userspace API for KUnit Fuzzing Harnesses

```
#define KFUZZ_IOC_MAGIC 'K'
#define KFUZZ_IOC_RUN _IO(KFUZZ_IOC_MAGIC, 1)

int main(void) {
    const char data[] = "Hello, world!";
    int fd = open("/sys/kernel/debug/kunit/example/fuzz", O_WRONLY, 0644);

    write(fd, data, sizeof(data));

    // One ID per fuzzing harness in the test suite.
    unsigned int handler_id = 1;
    if (ioctl(fd, KFUZZ_IOC_RUN, handler_id) < 0)
        return 1;

    close(fd);
    return 0;
}
```