CodeQL + DTrace = 💧🐞 in XNU

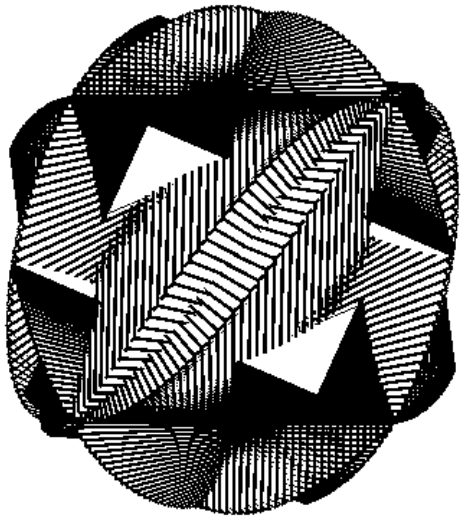
How to find multiple memory disclosures in XNU using CodeQL

ZeroCon

whoami

Arsenii Kostromin

- Security researcher
  - Focus on macOS security: userland and kernel
- Twitter @0x3C3E



Agenda

Kernel Memory Disclosure, my 🥇 and 🥈 bugs in XNU

Motivation

Apple interviewer asked me several times why I don't look for bugs in the kernel

- `Is it hard for you?`
- Before `December 2022`, I haven't looked into the `XNU` source code


My guiding moonlight...

Kernel Memory Disclosure

My approach

- Search online and tag writeups
- Prepare a debugging environment
- Use CodeQL to search for some patterns

Some `easy` bugs in XNU

- A tale of a simple Apple kernel bug
  - Weggli was used to find a specific pattern
- Finding a memory exposure vulnerability with CodeQL
  - CodeQL was used, the author found a bug in the DTrace module of XNU

How to debug kernel on a single M1 laptop?

- QEMU emulates Intel-based macOS
- DTrace, dynamic tracing framework in XNU

DTrace

- Released in `2005` by Oracle
- Apple merged it into XNU in `2007`
  - Was it thoroughly audited?
- It's complex and has its emulator in `the kernel`

```
#define DIF_OP_OR       1                   /* or   r1, r2, rd */
#define DIF_OP_XOR      2                   /* xor  r1, r2, rd */
...
#define DIF_OP_STRIP    80                  /* strip r1, key, rd */
```

bsd/sys/dtrace.h

CodeQL

- Framework for doing static analysis
- Models code as data → database
- Write logic-based SQL-like queries to find patterns

Building a CodeQL database

- Have to compile the program we want to query
- By default, some files were missing
- A great script to build a CodeQL database for XNU by pwn0rz

Code pattern

I decided to look for OOB issues. For that, I wrote a query to find such code, which meets the conditions below:

- `a >= b`, where `a` is signed, and `b` is not
- No `a <= 0` and `a < 0` checks
- `a` is an array index

`a >= b`, where `a` is signed, and `b` is not

```
from Variable arg
where exists(
    GEExpr ge | ge.getLeftOperand() = arg.getAnAccess()
    and ge.getLeftOperand().
      getExplicitlyConverted().
      getUnderlyingType().(IntegralType).isSigned()
    and ge.getRightOperand().
      getExplicitlyConverted().
      getUnderlyingType().(IntegralType).isUnsigned()
  )
select arg
```

No `a < 0` and `a <= 0` checks

```
from Variable arg
where not exists(
    LTExpr le | le.getLeftOperand() = arg.getAnAccess()
    and le.getRightOperand().getValue() = "0"
  )
  and not exists(
    LEExpr le | le.getLeftOperand() = arg.getAnAccess()
    and le.getRightOperand().getValue() = "0"
  )
select arg
```

`a` is an array index

```
from Variable arg, ArrayExpr ae
where ae.getArrayOffset() = arg.getAnAccess()
select ae.getArrayOffset(),
       ae.getEnclosingFunction()
```

Combined

```
from Variable arg, ArrayExpr ae
where exists(
    GEExpr ge | ge.getLeftOperand() = arg.getAnAccess()
    and ge.getLeftOperand().
      getExplicitlyConverted().
      getUnderlyingType().(IntegralType).isSigned()
    and ge.getRightOperand().
      getExplicitlyConverted().
      getUnderlyingType().(IntegralType).isUnsigned()
  )
  and not exists(
    LTExpr le | le.getLeftOperand() = arg.getAnAccess()
    and le.getRightOperand().getValue() = "0"
  )
  and not exists(
    LEExpr le | le.getLeftOperand() = arg.getAnAccess()
    and le.getRightOperand().getValue() = "0"
  )
  and ae.getArrayOffset() = arg.getAnAccess()
select ae.getArrayOffset(),
       ae.getEnclosingFunction()
```

The query produces

- `20` results
- Only `6` different functions

fasttrap_pid_getargdesc

```
// args: (void *arg, dtrace_id_t id, void *parg, dtrace_argdesc_t *desc)
if (probe->ftp_prov->ftp_retired != 0 ||
    desc->dtargd_ndx >= probe->ftp_nargs) {
  desc->dtargd_ndx = DTRACE_ARGNONE;
  return;
}

ndx = (probe->ftp_argmap != NULL) ?
  probe->ftp_argmap[desc->dtargd_ndx] : desc->dtargd_ndx;
```

Docs: get the argument description for args[X]

bsd/dev/dtrace/fasttrap.c

`dtargd_ndx` is `int`

```
typedef struct dtrace_argdesc {
...
        int dtargd_ndx;                         /* arg number (-1 iff none) */
...
} dtrace_argdesc_t;
```

`ftp_nargs` is `unsigned char`

```
struct fasttrap_probe {
...
        uint8_t ftp_nargs;                      /* translated argument count */
...
};
```

bsd/sys/dtrace.h, bsd/sys/fasttrap_impl.h

Both sides are converted to `int`

As `desc->dtargd_ndx` is `int` and `probe->ftp_nargs` is `unsigned char`

```
if (probe->ftp_prov->ftp_retired != 0 ||
    desc->dtargd_ndx >= probe->ftp_nargs) {
  desc->dtargd_ndx = DTRACE_ARGNONE;
  return;
}
```

If `desc->dtargd_ndx < 0`, then `desc->dtargd_ndx >= probe->ftp_nargs` is always `false`

⚔️ OOB Read, `desc->dtargd_ndx` is an index

```
ndx = (probe->ftp_argmap != NULL) ?
  probe->ftp_argmap[desc->dtargd_ndx] : desc->dtargd_ndx;
```

If `probe->ftp_argmap` isn't `null`, it's possible to reach the first expression and use `desc->dtargd_ndx` with values less than `0`

No direct calls to the function

It's called as a C-style `virtual function`

dtrace_pops

```
typedef struct dtrace_pops {
...
        void (*dtps_getargdesc)(void *arg, dtrace_id_t id, void *parg,
            dtrace_argdesc_t *desc);
...
} dtrace_pops_t;
```

dtrace_pops_t

```
static dtrace_pops_t pid_pops = {
...
        .dtps_getargdesc =      fasttrap_pid_getargdesc,
...
};
```

bsd/sys/dtrace.h, bsd/dev/dtrace/fasttrap.c

`dtps_getargdesc` might be a pointer to `fasttrap_pid_getargdesc`

```
prov->dtpv_pops.dtps_getargdesc(
    prov->dtpv_arg,
    probe->dtpr_id,
    probe->dtpr_arg,
    &desc
);
```

bsd/dev/dtrace/dtrace.c

Upper bound check in `fasttrap_pid_getargdesc`

```
if (probe->ftp_prov->ftp_retired != 0 ||
    desc->dtargd_ndx >= probe->ftp_nargs) {
  desc->dtargd_ndx = DTRACE_ARGNONE;
  return;
}
```

Comparing to `-1` in `dtrace_ioctl`

```
if (desc.dtargd_ndx == DTRACE_ARGNONE)
  return (EINVAL);
```

bsd/dev/dtrace/fasttrap.c, bsd/dev/dtrace/dtrace.c

How to leak out-of-bounds values?

```
ndx = (probe->ftp_argmap != NULL) ?
  probe->ftp_argmap[desc->dtargd_ndx] : desc->dtargd_ndx;

str = probe->ftp_ntypes;
for (i = 0; i < ndx; i++) {
  str += strlen(str) + 1;
}

(void) strlcpy(desc->dtargd_native, str, sizeof(desc->dtargd_native));
```

- We control integer index `desc->dtargd_ndx` and array of `null` delimited strings `probe->ftp_ntypes` (array of chars)
- We have to leak `probe->ftp_argmap[desc->dtargd_ndx]` (`ndx` is integer) value into `desc->dtargd_native`

The idea

```
str = probe->ftp_ntypes; // { 1, 1, 0, 1, 0, 2, 0, 3, 0, ...}
for (i = 0; i < ndx; i++) { // ndx is a value to leak
  str += strlen(str) + 1;
}
(void) strlcpy(desc->dtargd_native, str, sizeof(desc->dtargd_native));
```

- We could populate `probe->ftp_ntypes` with an array of null delimited strings
  - `[1, 1, 0, 1, 0, 2, 0, 3, 0, ..., 255]` from 0 to 255 (showed as bytes)
  - Encode `0` for example as `[1, 1, 0]`, so it's copied to the userland
- Then `ndx` equals to value in `str`
  - Special case — `0` is `"\x01\x01\x00"`

ndx = 0

```
str = probe->ftp_ntypes;    // { 1, 1, 0, 1, 0, 2, 0, 3, 0, ...}
for (i = 0; i < ndx; i++) { //  ^
  str += strlen(str) + 1;
}
// str points to "\x01\x01\x00"
(void) strlcpy(desc->dtargd_native, str, sizeof(desc->dtargd_native));
```

ndx = 1

```
str = probe->ftp_ntypes;    // { 1, 1, 0, 1, 0, 2, 0, 3, 0, ...}
for (i = 0; i < ndx; i++) { //            ^
  str += strlen(str) + 1;
}
// str points to "\x01\x00"
(void) strlcpy(desc->dtargd_native, str, sizeof(desc->dtargd_native));
```

How to reach?

`_dtrace_ioctl` → `DTRACEIOC_PROBEARG` switch case → `fasttrap_pid_getargdesc`

CVE-2023-27941

Kernel

`Available for: macOS Ventura`

`Impact: An app may be able to disclose kernel memory`

`Description: An out-of-bounds read issue existed that led to the disclosure of kernel memory. This was addressed with improved input validation.`

Details

- The bug allows reading data byte by byte in a range of 2GB
- Requires root access

Patch

Reversed `fasttrap_pid_getargdesc` changes

```
if (probe->ftp_prov->ftp_retired != 0 ||
    desc->dtargd_ndx < 0 || // added
    desc->dtargd_ndx >= probe->ftp_nargs) {
    desc->dtargd_ndx = DTRACE_ARGNONE;
    return;
}
```

- Apple hasn't released the new `XNU` source code

Kernel Memory Disclosure 2

Code pattern

- `a < b`, where `a` is signed
- The comparison above happens in `IfStmt`
- No `a <= 0` and `a < 0` checks
- `a` is an array index

`a < b`, where `a` is signed, happens in `IfStmt`

```
from Variable arg
where exists(
    LTExpr le |
    le.getLeftOperand() = arg.getAnAccess()
    and le.getParent() instanceof IfStmt
    and le.getLeftOperand().
        getExplicitlyConverted().
        getUnderlyingType().(IntegralType).isSigned()
  )
select arg
```

`IfStmt` is `if (a < b) {}`, but not `a < b` in `for (a = 0; a < b; a++)`

No `a < 0` and `a <= 0` checks

```
from Variable arg
where not exists(
    LTExpr le | le.getLeftOperand() = arg.getAnAccess()
    and le.getRightOperand().getValue() = "0"
  )
  and not exists(
    LEExpr le | le.getLeftOperand() = arg.getAnAccess()
    and le.getRightOperand().getValue() = "0"
  )
select arg
```

`a` is an array index

```
from Variable arg, ArrayExpr ae
where ae.getArrayOffset() = arg.getAnAccess()
select ae.getArrayOffset(),
       ae.getEnclosingFunction()
```

Filter results by a file path

```
from ArrayExpr ae
where ae.getFile().getAbsolutePath().
      matches("%/xnu-build/xnu/%")
  and not ae.getFile().getAbsolutePath().
      matches("%/xnu-build/xnu/SETUP/%")
select ae.getArrayOffset(),
       ae.getEnclosingFunction()
```

Combined

```
from Variable arg, ArrayExpr ae
where exists(
    LTExpr le |
    le.getLeftOperand() = arg.getAnAccess()
    and le.getParent() instanceof IfStmt
    and le.getLeftOperand().
        getExplicitlyConverted().
        getUnderlyingType().(IntegralType).isSigned()
  )
  and not exists(
    LTExpr le | le.getLeftOperand() = arg.getAnAccess()
    and le.getRightOperand().getValue() = "0"
  )
  and not exists(
    LEExpr le | le.getLeftOperand() = arg.getAnAccess()
    and le.getRightOperand().getValue() = "0"
  )
  and ae.getArrayOffset() = arg.getAnAccess()
  and ae.getFile().getAbsolutePath().matches("%/xnu-build/xnu/%")
  and not ae.getFile().getAbsolutePath().matches("%/xnu-build/xnu/SETUP/%")
select ae.getArrayOffset(),
       ae.getEnclosingFunction()
```

The query produces

- `169` results
- Only `45` different functions

⚔️ OOB Read, `argno` is an index on `arm64`

```
uint64_t
fasttrap_pid_getarg(void *arg, dtrace_id_t id, void *parg, int argno,
    int aframes)
{
        arm_saved_state_t* regs = find_user_regs(current_thread());

        /* First eight arguments are in registers */
        if (argno < 8) {
                return saved_state64(regs)->x[argno];
        }
```

Docs: get the value for an argX or args[X] variable

bsd/dev/arm64/fasttrap_isa.c

⚔️ OOB Read, `argno` is an index on `x86_64`

```
uint64_t
fasttrap_pid_getarg(void* arg, dtrace_id_t id, void* parg, int argno,
    int aframes)
{
    pal_register_cache_state(current_thread(), VALID);
    return (fasttrap_anarg(
        (x86_saved_state_t*)find_user_regs(current_thread()),
        1,
        argno));
}
```

`fasttrap_anarg`

```
// args: (x86_saved_state_t *regs, int function_entry, int argno)
if (argno < 6)
  return ((&regs64->rdi)[argno]);
```

bsd/dev/i386/fasttrap_isa.c, bsd/dev/i386/fasttrap_isa.c

dtrace_pops

```
typedef struct dtrace_pops {
...
        uint64_t (*dtps_getargval)(void *arg, dtrace_id_t id, void *parg,
            int argno, int aframes);
...
} dtrace_pops_t;
```

dtrace_pops_t

```
static dtrace_pops_t pid_pops = {
...
        .dtps_getargval =       fasttrap_pid_getarg,
...
};
```

bsd/dev/dtrace/fasttrap.c

`dtps_getargval` might be a pointer to `fasttrap_pid_getarg`

```
// func: dtrace_dif_variable
// args: (dtrace_mstate_t *mstate, dtrace_state_t *state, uint64_t v,
// uint64_t ndx)
val = pv->dtpv_pops.dtps_getargval(pv->dtpv_arg,
    mstate->dtms_probe->dtpr_id,
    mstate->dtms_probe->dtpr_arg, ndx, aframes);
```

bsd/dev/dtrace/dtrace.c

Bounds check?

```
// func: dtrace_dif_variable
// args: (dtrace_mstate_t *mstate, dtrace_state_t *state, uint64_t v,
// uint64_t ndx)
if (ndx >= sizeof (mstate->dtms_arg) / sizeof (mstate->dtms_arg[0])) {
...
  dtrace_provider_t *pv;
  uint64_t val;

  pv = mstate->dtms_probe->dtpr_provider;
  if (pv->dtpv_pops.dtps_getargval != NULL)
    val = pv->dtpv_pops.dtps_getargval(pv->dtpv_arg,
        mstate->dtms_probe->dtpr_id,
        mstate->dtms_probe->dtpr_arg, ndx, aframes);
```

`ndx` is an `unsigned long long`, later it's converted into an `int` in `fasttrap_pid_getarg`, `argno` argument

How to reach?

`dtrace_dif_emulate` → `DIF_OP_LDGA` opcode → `dtrace_dif_variable` → `fasttrap_pid_getarg`

An old PoC helped to trigger the vulnerable function

Almost the same code flow as in CVE-2017-13782 by Kevin Backhouse

- But you have to use a `fasttrap` provider, which allows tracing userland functions
  - It's possible to define a function `void foo() {}`
  - Trace it using DTrace: `pid$target::foo:entry { ... }`

Code flow difference

```
pv = mstate->dtms_probe->dtpr_provider;
if (pv->dtpv_pops.dtps_getargval != NULL)
    val = pv->dtpv_pops.dtps_getargval(pv->dtpv_arg,
        mstate->dtms_probe->dtpr_id,
        mstate->dtms_probe->dtpr_arg, ndx, aframes);   // CVE-2023-28200
...
else
    val = dtrace_getarg(ndx, aframes, mstate, vstate); // CVE-2017-13782
```

- `9` lines difference

bsd/dev/dtrace/dtrace.c

CVE-2023-28200

Kernel

`Available for: macOS Ventura`

`Impact: An app may be able to disclose kernel memory`

`Description: A validation issue was addressed with improved input sanitization.`

Details

- The bug allows reading data in a range of 16GB
- Requires root access

Patch

Reversed `dtrace_dif_variable` changes

```
if (ndx >= sizeof (mstate->dtms_arg) / sizeof (mstate->dtms_arg[0])) {
  if ((ndx & 0x80000000) != 0) return 0; // added
  ...
  dtrace_provider_t *pv;
  uint64_t val;

  pv = mstate->dtms_probe->dtpr_provider;
  if (pv->dtpv_pops.dtps_getargval != NULL)
    val = pv->dtpv_pops.dtps_getargval(pv->dtpv_arg,
        mstate->dtms_probe->dtpr_id,
        mstate->dtms_probe->dtpr_arg, ndx, aframes);
```

- Additional check added in caller function
- Callee functions are unfixed for some reason

Tweet

You Retweeted

John Aakerblom
@jaakerblom

Apple is choosing not to pay bounties for kernel infoleaks. Being in more demand than ever after new mitigations, I expect Apple to change their stance for such bugs reported by @antoniozekic, myself and others. Until then I recommend researchers present such bugs at @0x41con

21:51 · 31.08.2022

6 Retweets **1** Quote Tweet **67** Likes

Why?

- `root` access != `kernel` access on macOS
- `SIP` puts the whole system into a sandbox
  - even `root` can't load untrusted kernel extensions
- + I had `App Sandbox Escape` → `user to root` LPE chain

PoCs

- CVE-2023-27941 matches kernel addresses from leaked data
- CVE-2023-28200 only panics the kernel

Conclusion

- Apple has to maintain two architectures: `x86_64` and `arm64`
- C-like `virtual functions` make `static` analysis harder

Resources

- Real hackers don't leave DTrace
- Finding a memory exposure vulnerability with CodeQL
- There is no S in macOS SIP

Thank you

Q&A