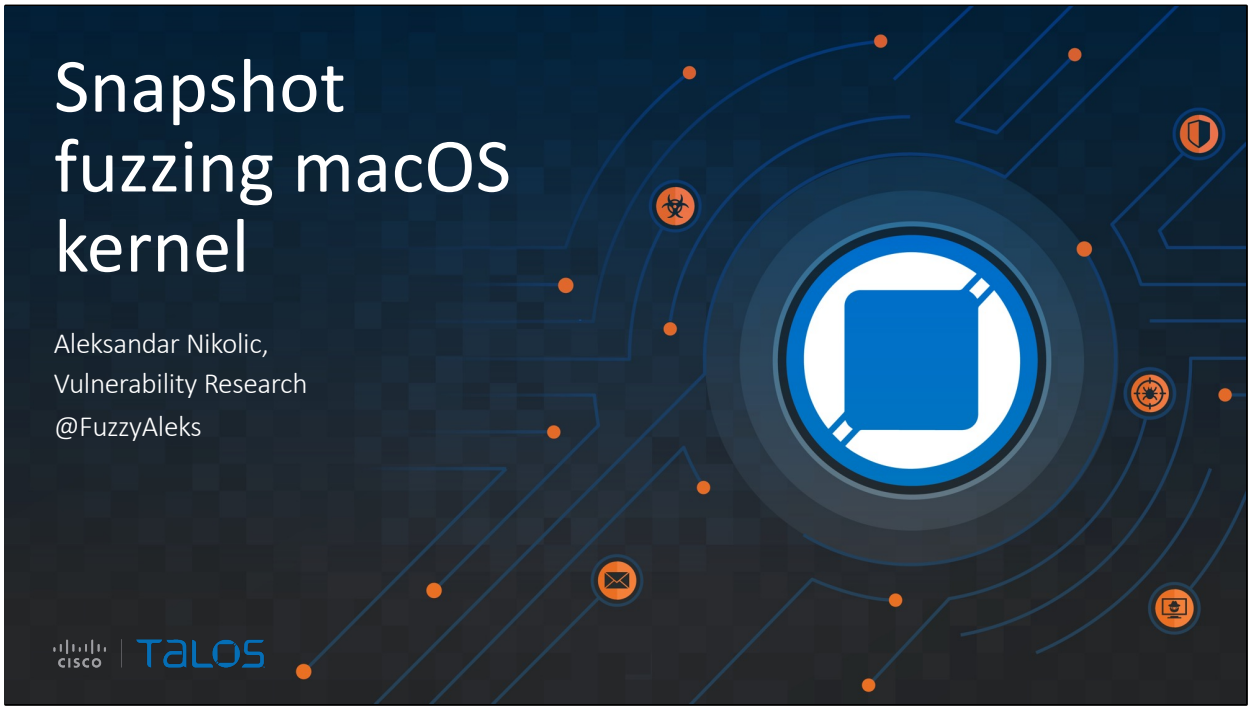
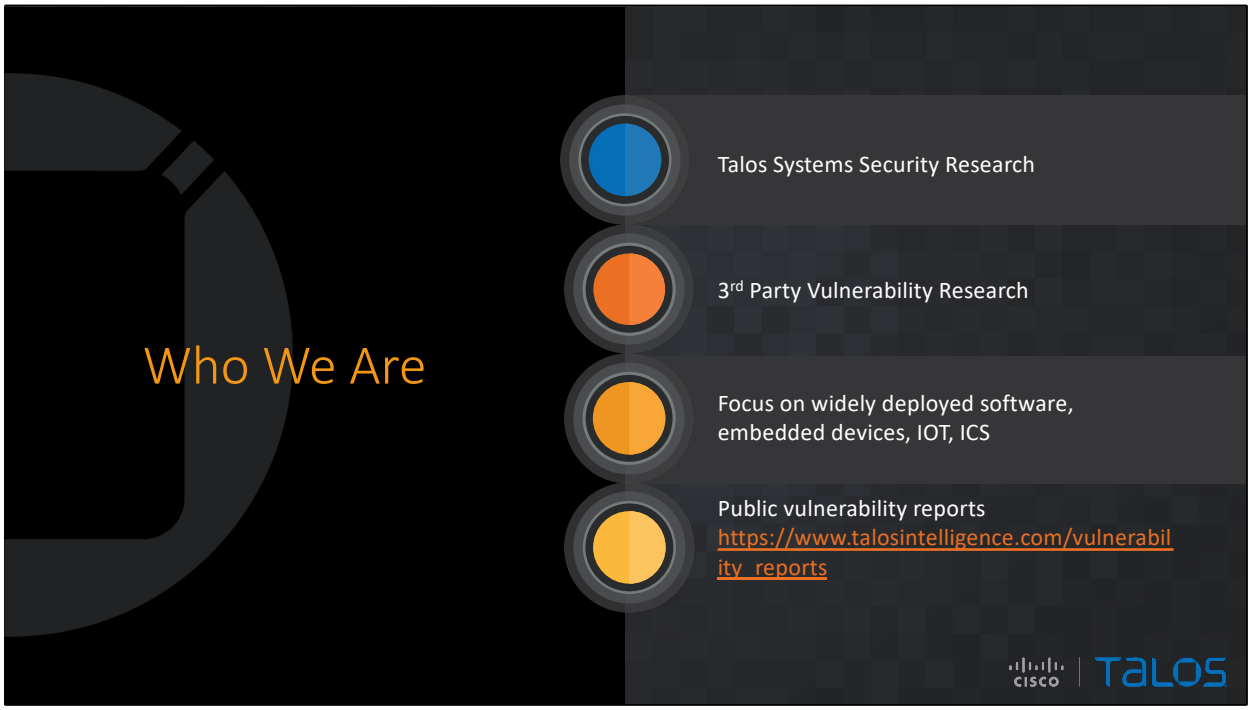
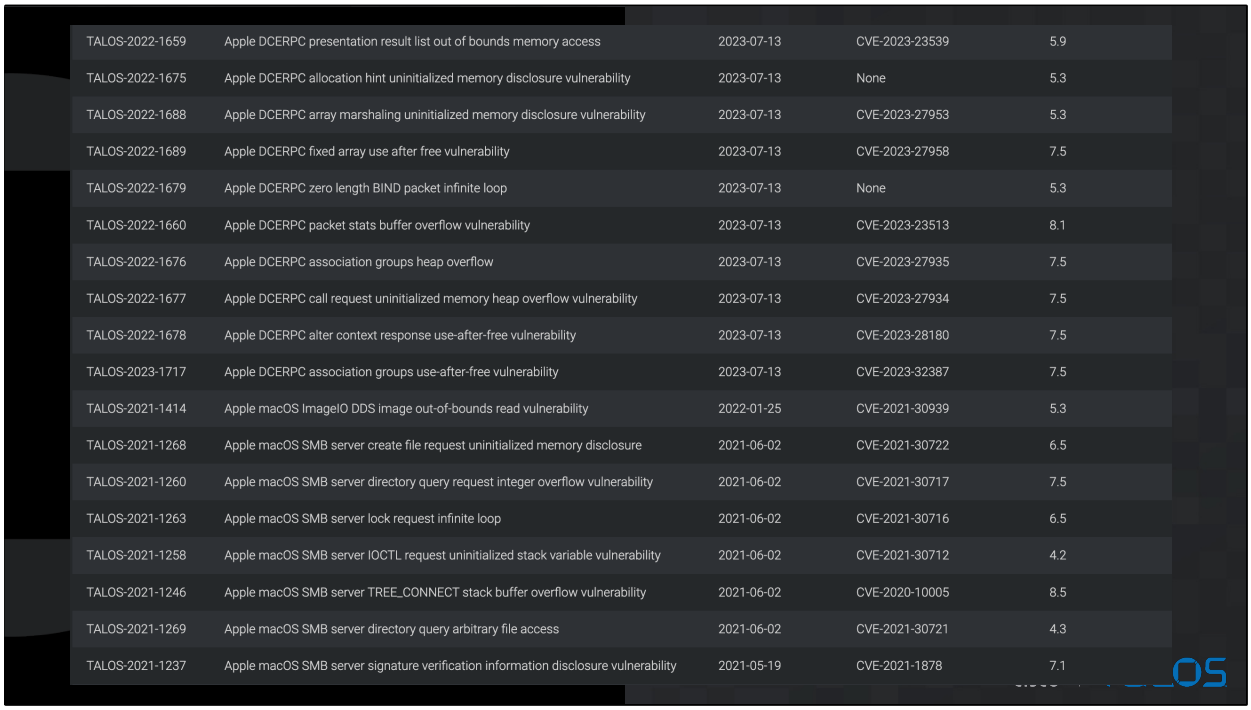


Hello everybody, today I am going to present our ongoing project that brings snapshot-based / coverage-guided fuzzing to macOS kernel

This project doesn't have a working codename.
It's been a tradition in macOS vulnerability research to title your work with some sort of an apple pun. Eating an apple, an apple core, apple seeds … it's all been done and I am open for suggestions.

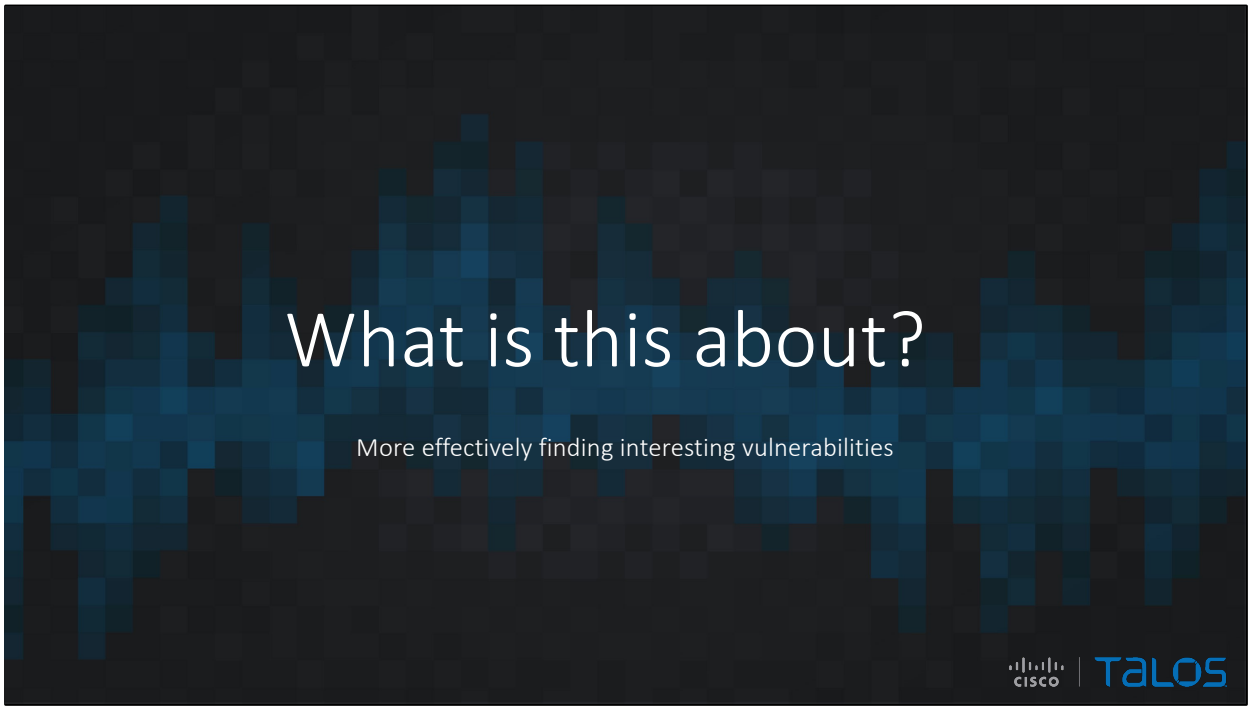


I'm going to breeze through my motivation for this work here. I am part of a vulnerability research team in Cisco Talos. Our one major task so to perform 3rd party vulnerability research.
Most of our work culminates in vulnerabilities getting fixed, detections for those vulnerabilities being in place and finally a public disclosure of the vulnerability through advisories.

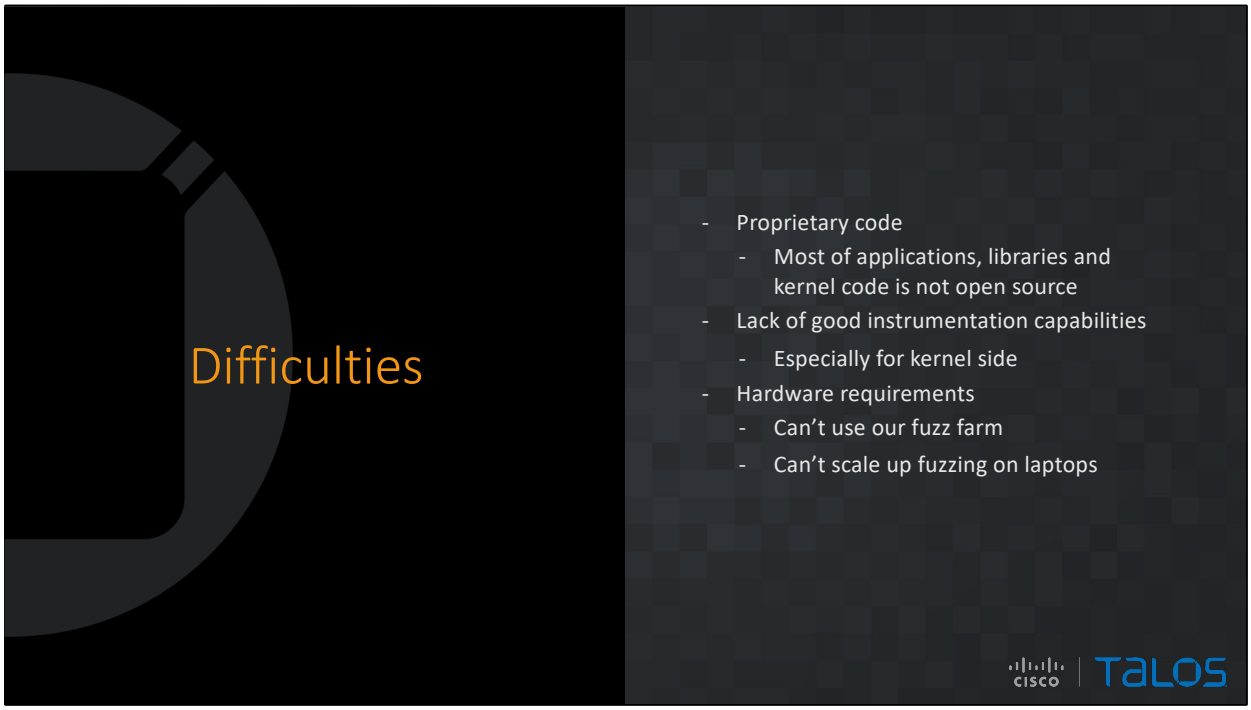


TALOS-2022-1659	Apple DCERPC presentation result list out of bounds memory access	2023-07-13	CVE-2023-23539	5.9
TALOS-2022-1675	Apple DCERPC allocation hint uninitialized memory disclosure vulnerability	2023-07-13	None	5.3
TALOS-2022-1688	Apple DCERPC array marshaling uninitialized memory disclosure vulnerability	2023-07-13	CVE-2023-27953	5.3
TALOS-2022-1689	Apple DCERPC fixed array use after free vulnerability	2023-07-13	CVE-2023-27958	7.5
TALOS-2022-1679	Apple DCERPC zero length BIND packet infinite loop	2023-07-13	None	5.3
TALOS-2022-1660	Apple DCERPC packet stats buffer overflow vulnerability	2023-07-13	CVE-2023-23513	8.1
TALOS-2022-1676	Apple DCERPC association groups heap overflow	2023-07-13	CVE-2023-27935	7.5
TALOS-2022-1677	Apple DCERPC call request uninitialized memory heap overflow vulnerability	2023-07-13	CVE-2023-27934	7.5
TALOS-2022-1678	Apple DCERPC alter context response use-after-free vulnerability	2023-07-13	CVE-2023-28180	7.5
TALOS-2023-1717	Apple DCERPC association groups use-after-free vulnerability	2023-07-13	CVE-2023-32387	7.5
TALOS-2021-1414	Apple macOS ImageIO DDS image out-of-bounds read vulnerability	2022-01-25	CVE-2021-30939	5.3
TALOS-2021-1268	Apple macOS SMB server create file request uninitialized memory disclosure	2021-06-02	CVE-2021-30722	6.5
TALOS-2021-1260	Apple macOS SMB server directory query request integer overflow vulnerability	2021-06-02	CVE-2021-30717	7.5
TALOS-2021-1263	Apple macOS SMB server lock request infinite loop	2021-06-02	CVE-2021-30716	6.5
TALOS-2021-1258	Apple macOS SMB server IOCTL request uninitialized stack variable vulnerability	2021-06-02	CVE-2021-30712	4.2
TALOS-2021-1246	Apple macOS SMB server TREE_CONNECT stack buffer overflow vulnerability	2021-06-02	CVE-2020-10005	8.5
TALOS-2021-1269	Apple macOS SMB server directory query arbitrary file access	2021-06-02	CVE-2021-30721	4.3
TALOS-2021-1237	Apple macOS SMB server signature verification information disclosure vulnerability	2021-05-19	CVE-2021-1878	7.1

We do report a sizeable amount of vulnerabilities to apple each year.



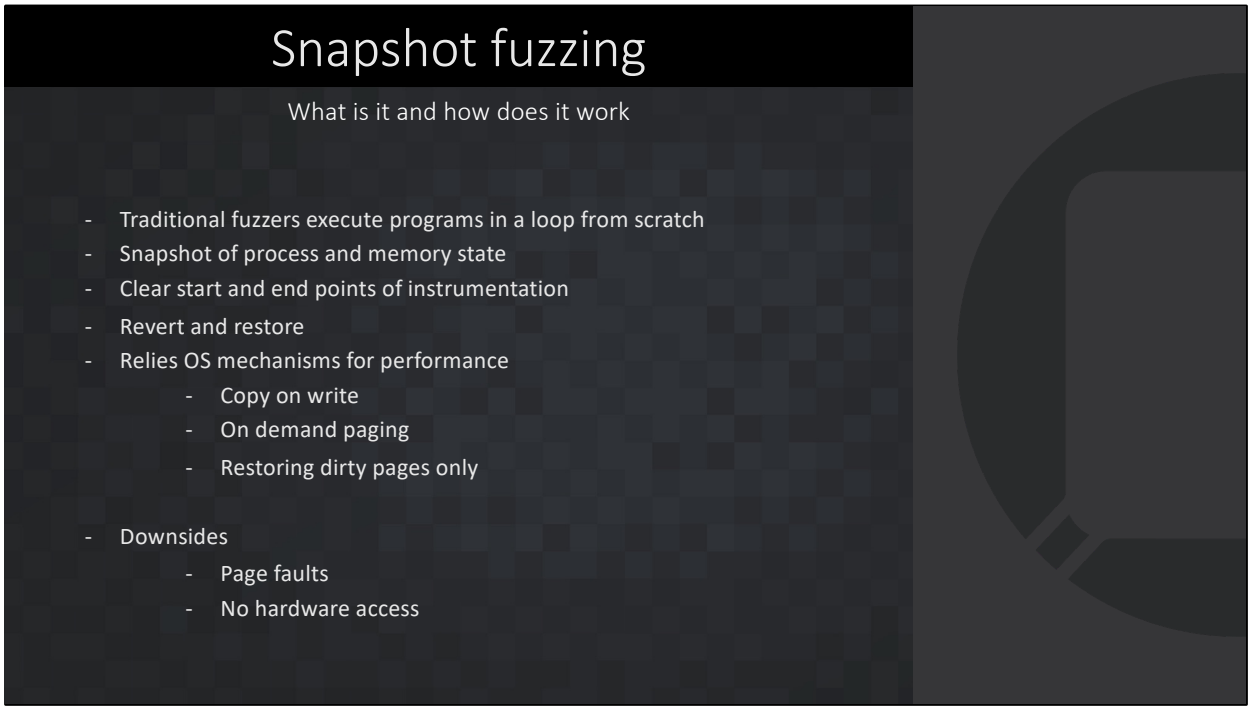
Finding vulnerabilities is our goal. Finding better ways of finding good vulnerabilities is what we work on. Some software is harder to test and analyze and requires better tooling than other.
I will walk you through our projects that enables us to effectively test macOS kernel comonents on commodity hardware with coverage analysis.



Compared to fuzzing everything on linux where most of the things are open source fuzzing anything on macOS presents a few difficulties. Things are closed source so we can't use compile time instrumentation. Dynamic Binary instrumentation Tools like Dynamorio and TinyInst work on macOS , but cannot be used to instrument kernel components.

And then , there are hardware requirements. With few exceptions, macOS only runs on apple hardware. Yes , it can be virtualized, but that has it's other drawbacks. What this means in practice is that we cannot use our commodity off the shelf servers to test macOS code, fuzzing on laptops isn't exactly effective.

This project aims to alleviate most of these issues. Using snapshot based approach enables us to precisely target closed source code without the need for custom harnesses. We get full instrumentation and code coverage by executing tests in an emulator which in turn enables us to perform tests on any hardware and to use any resources we have available.

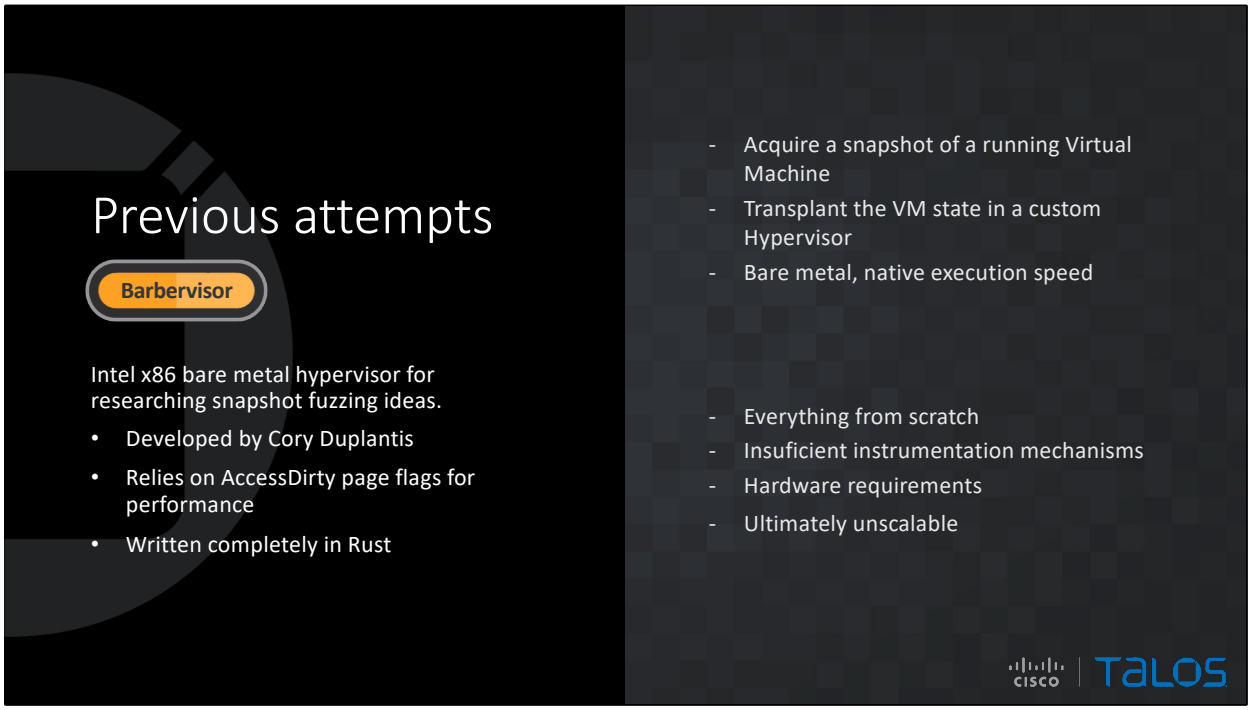


So what is snapshot based fuzzing?
The simplest way to fuzz a target application is to run it in a loop while changing the inputs. The obvious downside of this is that you lose time on application initialization , boilerplate code and overall you spend a lot of CPU time not actually executing the relevant part of the code.
Snapshot fuzzing is also sometimes called in-memory fuzzing.

The approach in snapshot based fuzzing is to define a point in process execution where you want to inject the fuzzing testcase (at an entry point of an important function) , you then interrupt the program at that point (via breakpoint or other means) and take a snapshot. Snapshot includes all of virtual memory being used as well as CPU or other process state that would be required to restore and resume process execution. Then you insert the fuzzing testcase by modifying the memory and resume execution.
When the execution reaches a predefined sink (end of function , error state , etc.) you stop the program, discard and replace the state with the previously saved one. Rinse and repeat .

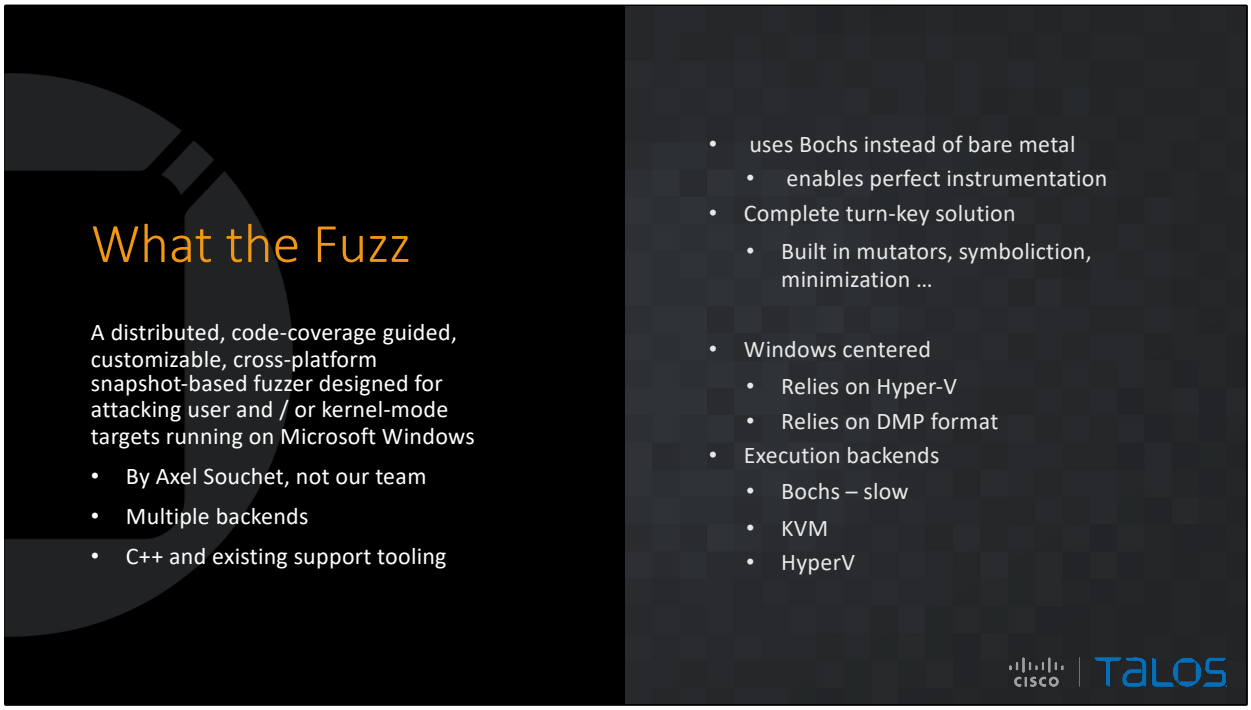
Benefit of this is that you only pay the penalty of restoring the process to previous

state, you don’t create it from scratch. Additionally, if you can rely on OS or CPU mechanisms such as CopyOnWrite , page dirty markings and on-demand paging, operation of restoring the process can be very fast and have very little impact on overall fuzzing speed.



Our previous attemts at utilizing snapshot based fuzzing where championed by Cory Duplantis in his work on Barbervisor which was a bare metal hypervisor developed for the sole purpose of supporting high performance snapshot fuzzing.
It involved acquiring a snapshot of a full (virtual box based) VM and then transplanting it into Barbervisor where it could be executed. It relied on Intel CPU features to enable high performance by only restoring modified memory pages.

While this showed great potential and gave us a glimpse of the utility that snapshot based fuzzing could provide, it had a few downsides.



Around the time Cory published Barbervisor, Axel published his WTF project which takes a different approach. It trades performance focus to have a nice development environment by relying on existing tooling. It uses Hyper-V to run virtual machines that are to be snapshotted, then uses kd (windows kernel debugger) to perform the snapshot which saves the state in a Windows memory dump file format which is actually optimized for loading already. Written in C++ which means it can benefit from the plethora of existing support libraries such as custom mutators or fuzz generators.

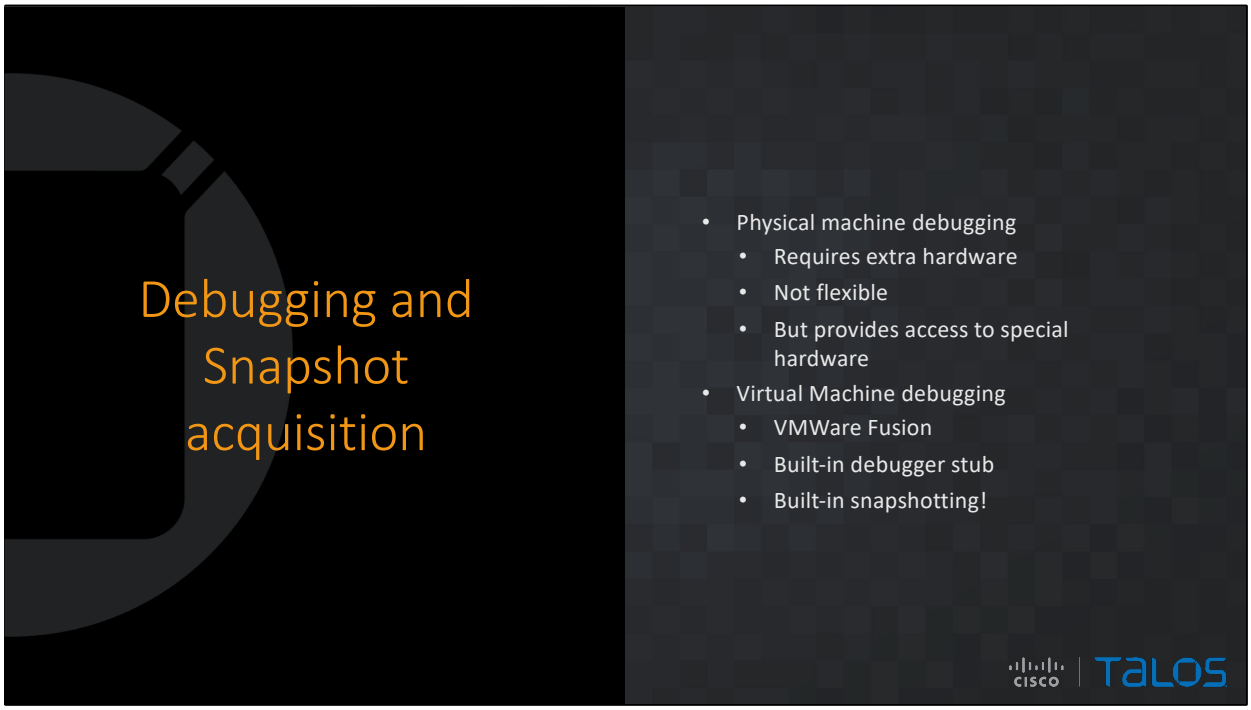
It has multiple possible execution backhends. Most full featured is based on Bochs , an x86 emulator , which provides complete instrumentation framework. You pay for it in performance, it's many times slower than native of course, but you can run it on any platform that Bochs runs on (linux , windows, virtualized or otherwise) which no special hardware requirements.

The biggest downside is that it can only target Windows virtual machines and targets running on Windows. And we want to fuzz macOS Kernel….

When it comes to modifying WTF to support fuzzing macOS targets we need to take care of a couple of mechanisms that don't support that. Split into pre-fuzzing and fuzzing stage those include:

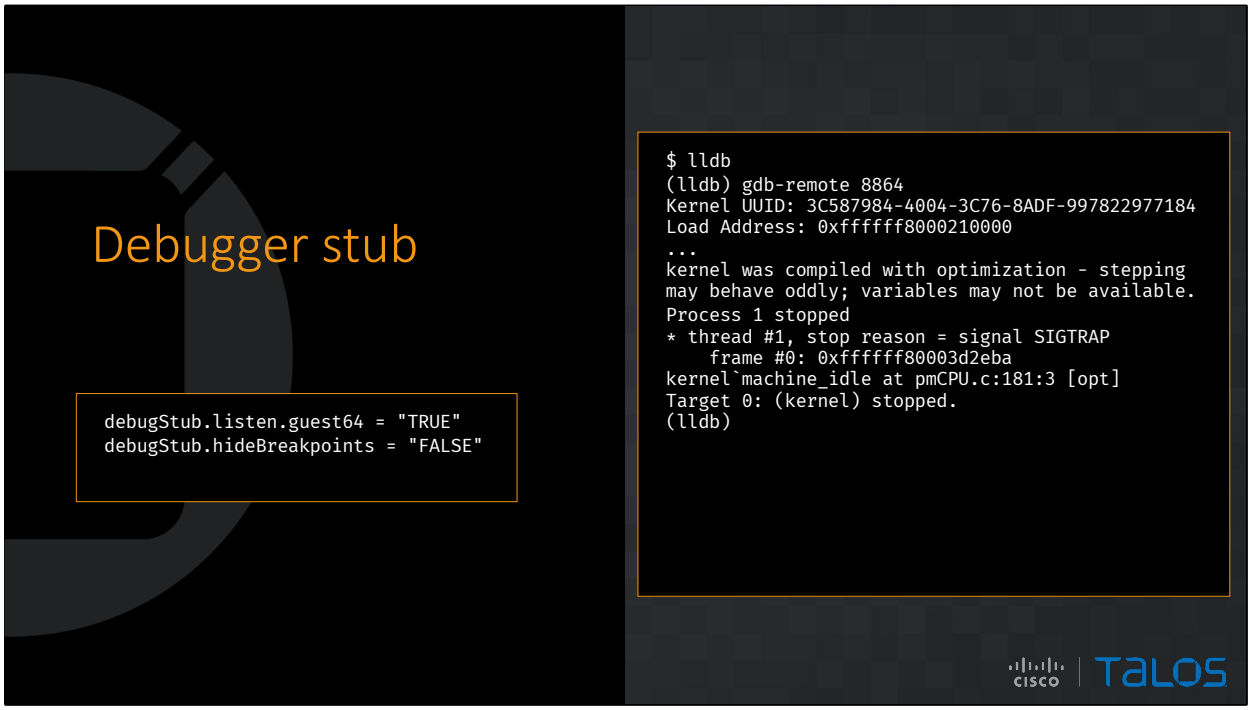
- A mechanism to debug the OS and process that is to be fuzzed – this is necessary to precisely choose the point of snapshotting
- A mechanism to acquire a copy of physical memory – obviously necessary in order to transplant the execution into the emulator
- CPU state snapshoting – this has to include all the Control Registers , all the MSRs and other CPU specific registers that aren't simple general purpose registers

- In the fuzzing stage on the other hand, we need a mechanism to restore the acquired memory pages – this has to be custom for our environment.
- We also need a way to catch crashes as crashing/faulting mechanisms on Windows and macOS differ a lot.
- CPU state, memory modification and coverage analysis will also require adjustments



For macOS kernel , Ideally , we'd want to be able to take a snapshot of an actual, physical machine. That would give us the most accurate attack surface with all the kernel extensions that require special hardware being loaded and set up. There is a significan attack surface reduction in virtualized macOS.
However, debugging physical mac machines is difficult. It requires at least one more machine , special network adapters and the debug mechanism isn't perfect for our goal (relies on non-maskable interrupts instead of breakpoints, and doesn't actually fully stop the kernel from executing code).

Debugging a virtual machine on the other hand is somewhat easier. VMWare Fusion contains a gdb server stub which you can enable that's doesn't care what the underlying operating system is. Also, we can piggyback on VMWare's snapshoting feature. So we went with the second option.

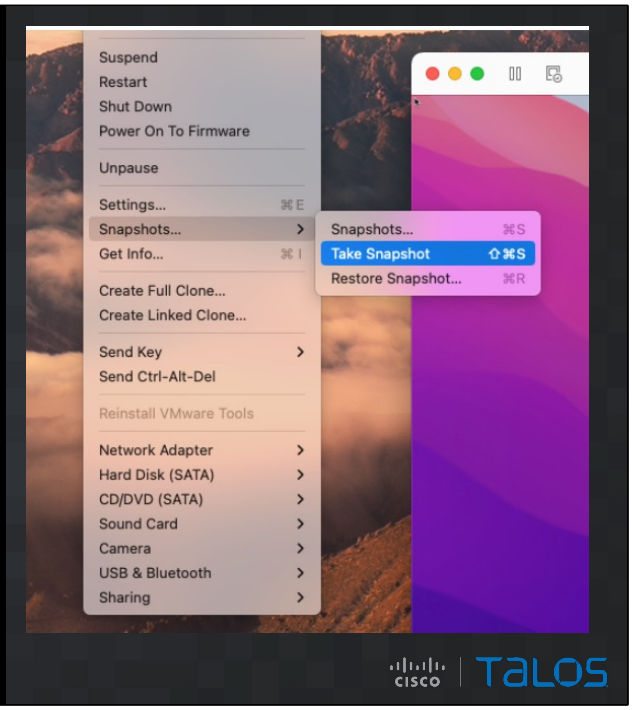


VMWare debugger stub is enabled in the vmx file like so. First option enables it and the second option tells gdb stub to simply use int3 as breakpoints which works perfectly fine.
On the left we can see how you attach to a running VM and break into it's kernel. Easy peasy.

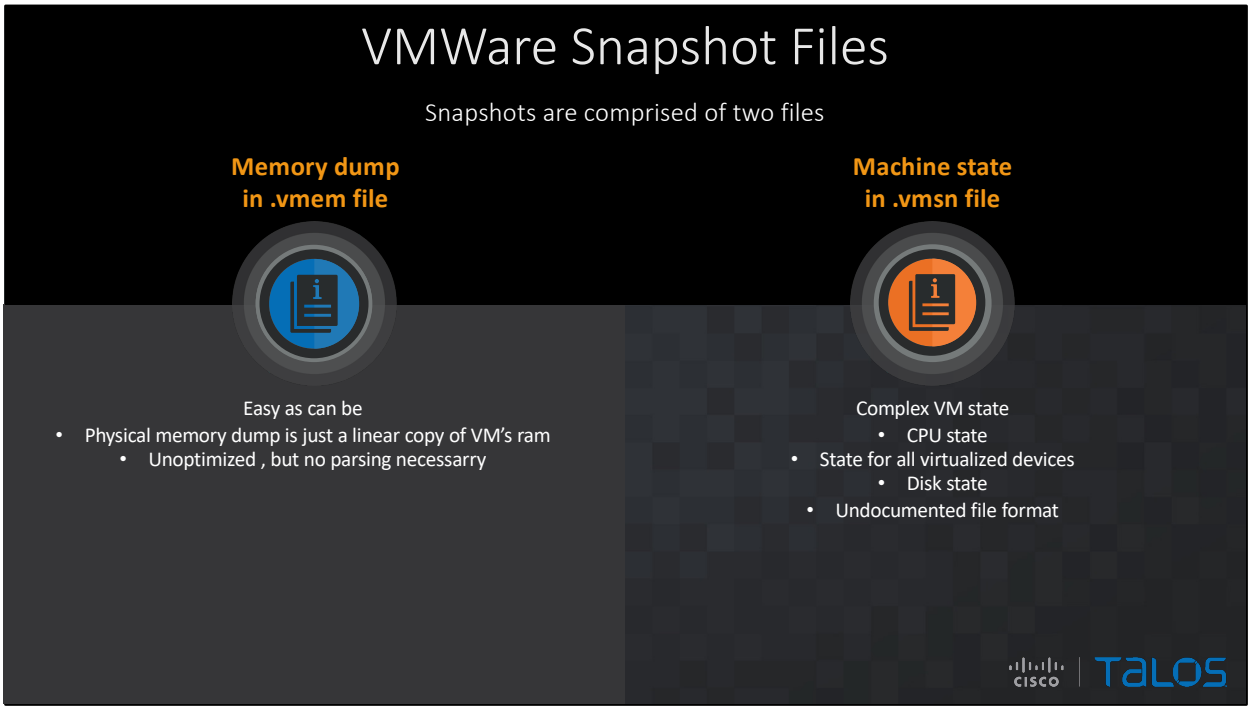
Snapshotting

VMWare's Snapshot/Restore mechanism does exactly what we want.

- Break in the debugger and make a snapshot



Second major requirement for snapshot fuzzing is, well, snapshotting. We can piggyback on VMWare Fusion for this as well.
The usual way to use VMWare's snapshotting is to either suspend a VM or make an exact copy you can revert to. This is almost exactly what we want to do.
We set a break point in the debugger, wait for it to be reached at which point execution is paused. Then we can simply take a snapshot of a very precise moment.
No need to time anything or guess when code is being executed. Since we are debugging the VM, we control it fully.
A slightly more difficult part is figuring out how to use this snapshot.

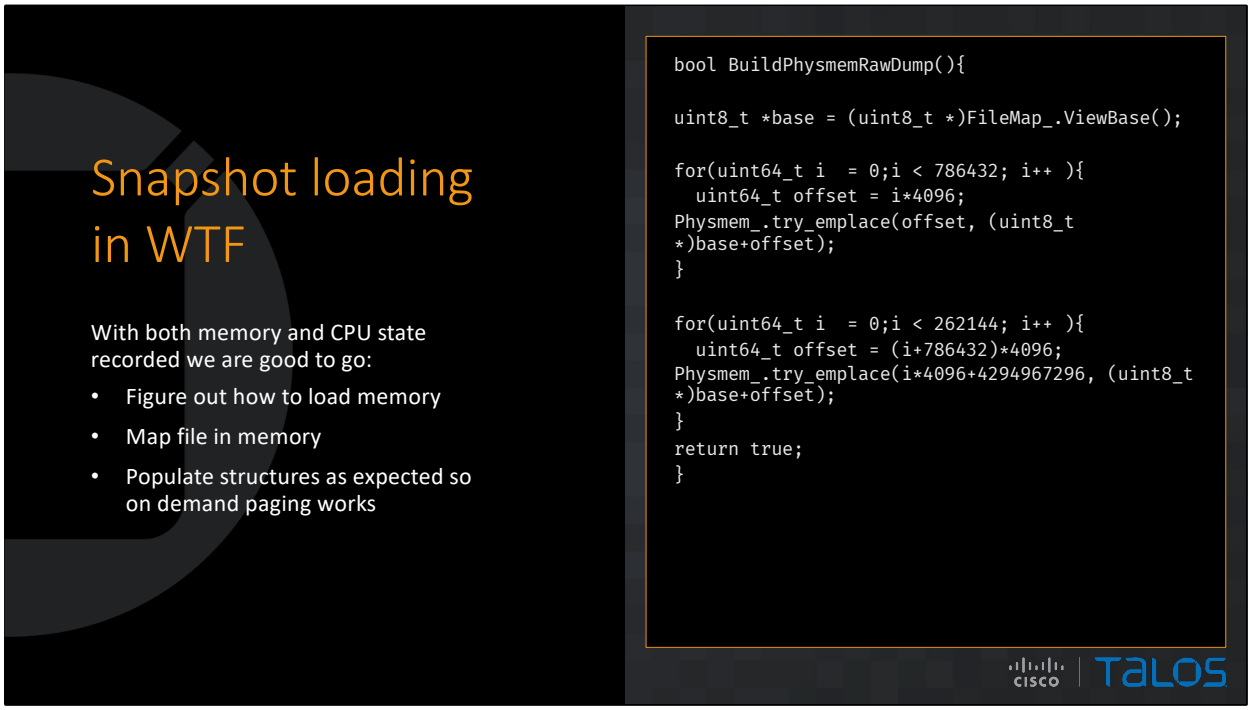


To be able to reuse them for our purposes we needed to figure out file formats of Vmware snapshots. Snapshots are essentially just two separate files. A vmem file which holds memory state and a vmsn file which holds device state which includes CPU, all the controlers , busses , pci , disks … Everything that's actually needed to restore the VM precisely.

As far as memory dump goes, we are in luck and the vmem file is simply a linear dump of all of VMs ram. If the vm has 2gb of ram , vmem file will be 2gb byte for byte copy of ram contents. This is physical memory layout of course, because we are dealing with virtual machines (I know, confusing a bit). So , no parsing is required , we'll just need a simpler loader.

Machine state file on the other hand is fairly complex , undocumented format that contains a lot of stuff we don't care about. We only care about CPU state as we won't be trying to restore a complete VM , just enough to run a fair bit of code.

Machine state file format is undocumented, but has been mostly reverse engineered for use in Volatility project. It was slightly broken for this version of VMWare Fusion and macOS guests in particular, but it was a great starting point. It was fairly easy to fix it and patch it so it spits out information that we need.
Here you can see a shortened output that includes necessary CPU state registers such as idtr , ldtr , etc… For actual fuzzing , we need a complete dump of all registers which this script does fully (they just wouldn’t fit on a slide).



With both file formats figured out, we can go back to WTF to modify it accordingly. The most important modification we need to make is to the physical memory loader. As mentioned, WTF uses Windows' dmp file format for this so we need our own handler. Since our memory dump file is just a direct 1 to 1 copy of physical ram, mapping it into memory and then mapping the pages is very straightforward as you can see. We just fake the structures with appropriate offsets and are good to go.

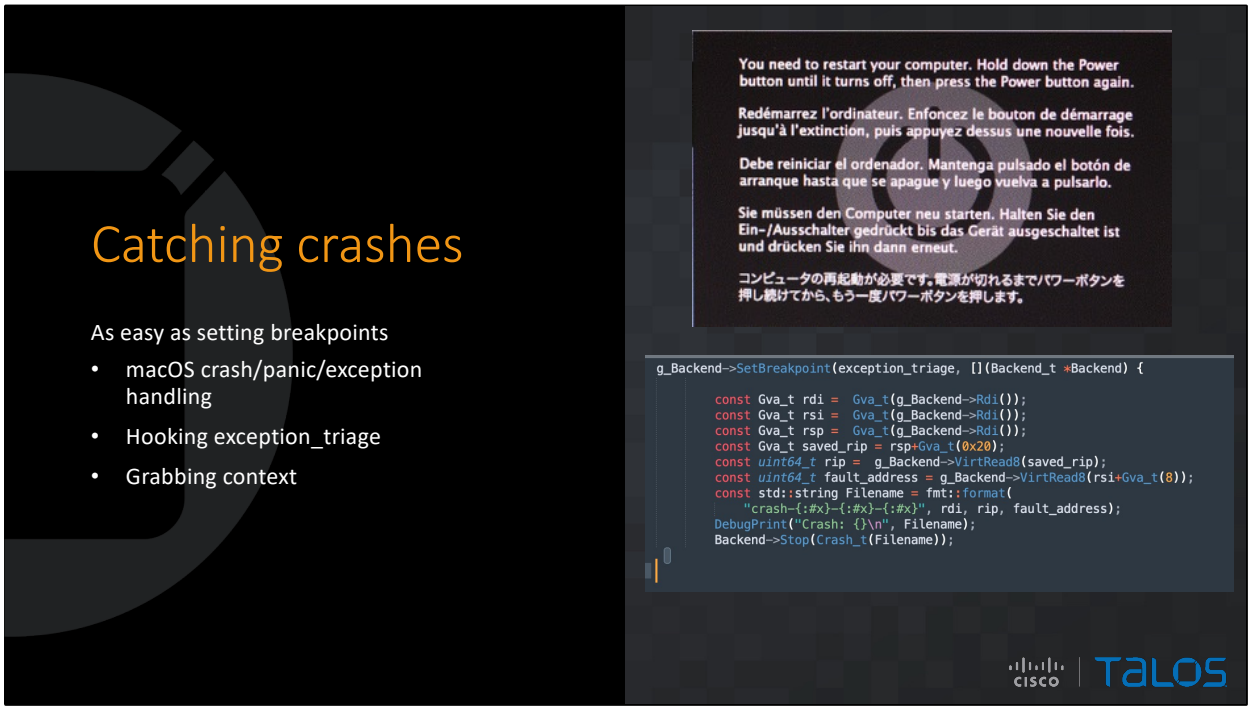
Good to go

With both CPU and memory in appropriate format , we can try it out

```
c:\work\codes\wtf\targets\ipv6_input>..\..\src\build\wtf.exe  run  --backend=bochscpu --name IPv6_Input --
state state --input inputs\ipv6 --trace-type 1 --trace-path .
The debugger instance is loaded with 0 items
load raw mem dump1
Done
Setting debug register status to zero.
Setting debug register status to zero.
Segment with selector 0 has invalid attributes.
Segment with selector 0 has invalid attributes.
Segment with selector 8 has invalid attributes.
Segment with selector 0 has invalid attributes.
Segment with selector 10 has invalid attributes.
Segment with selector 0 has invalid attributes.
Trace file .\ipv6.trace
Running inputs\ipv6
------------------------------------------------
Run stats:
Instructions executed: 13001 (4961 unique)
         Dirty pages: 229376 bytes (0 MB)
      Memory accesses: 46135 bytes (0 MB)
#1 cov: 4961 exec/s: infm lastcov: 0.0s crash: 0 timeout: 0 cr3: 0 uptime: 0.0s
```

After some inevitable debugging , we actually got the whole scheme to work! This shows an example of a snapshot created on macOS , taken with VMWare fusion, moved onto a Windows machine , loaded into WTF and resumed. Ignoring some of the debugging output, we can see that in this trial run it has executed a total of 13001 instruction in very little time!
So , that means we have captured a snapshot of a running macos VM , transplanted it onto whatever other platform (windows in this case) and continued to execute it there.

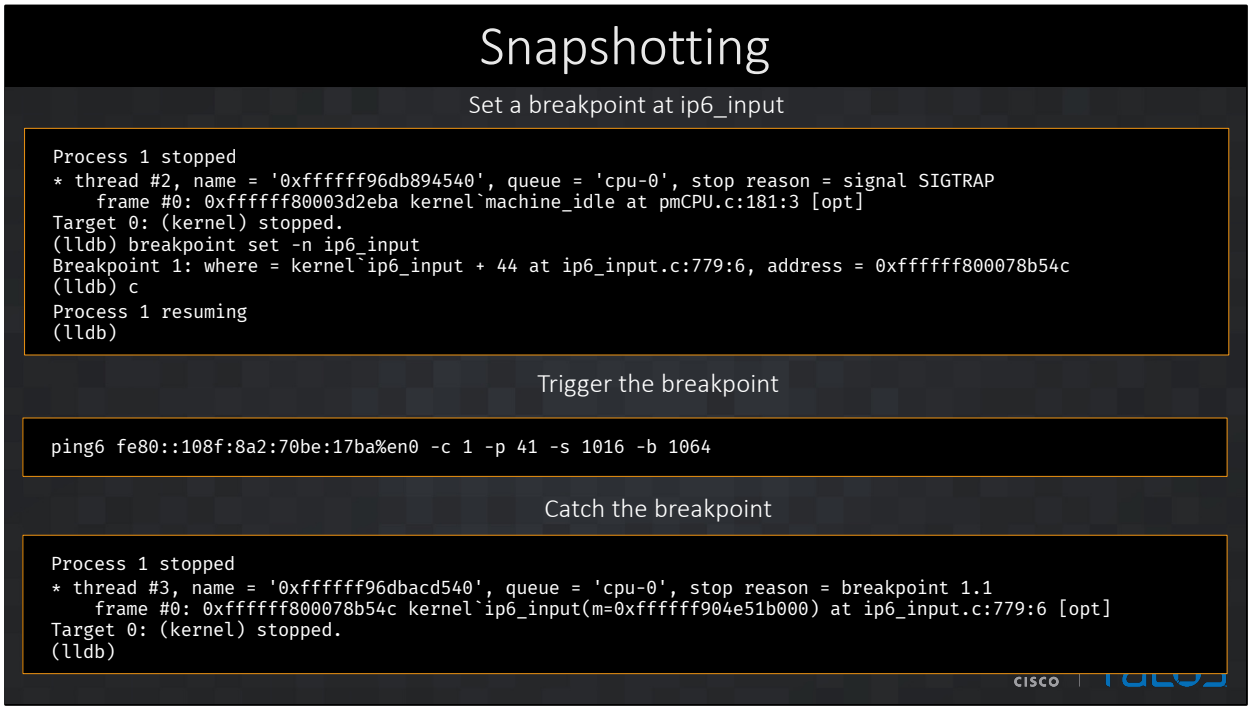
This is a simple example that just runs the code without modifying the input , but it does show that we can indeed run the code in an emulator with coverage.
What's left is just minor details…



Last piece of the puzzle is about how to actually catch crashes. In WTF and our modification of it, this is as simple as setting a breakpoint at an appropriate place. What is an appropriate place? Well, kernel panics, exceptions , faults etc. on macOS go through a complicated callstack that ultimately culminates in the above image or something similar.
Depending on what type of crash we are trying to catch, and the type of kernel we are running , we can simply put a breakpoint on `exception_triage` function which is in the execution path between a fault happening and machine panicing or rebooting.

Lets see how fuzzing a concrete example would look like. Lets say we want to target macOS' IPv6 stack. This is a simple but very interesting entrypoint into some complex code. Fairly complex set of protocols, reachable over network, etc.
It would be pretty difficult to fuzz with traditional fuzzers because network fuzzing is slow, and we wouldn't have coverage.

Our fuzzing target is function ip6_input which is actually the entry point for parsing incoming ipv6 packets. It has a single parameter which contains an mbuf that holds the actual packet data. This is the data that we want to mutate and modity in order to fuzz ipv6_input.
Mbufs are a fairly standard structure in XNU and essentially is a linked list of buffers that contain data. We just need to find where the actual packet data is (mh_data) and mutate it before resuming execution.



To create a snapshot , we use the debugger to set a breakpoint at ip6_input function. This is where we want to start our fuzzing.

Then, we need to provoke the VM to actually reach that breakpoint. We can either wait till the VM receives an ipv6 packet, or we can do it manually like with this ping6 command.

When ping6 comman is executed, the VM will receive the ip6 packet and will start parsing it which will immediately reach our breakpoint.
The VM is now paused and we have the address of our mbuf that contains the packet which we can fuzz.

Fuzzer harness

Set up catching crashes

```
Gva_t exception_triage = Gva_t(0xffffff8000283cb0);
  if (!g_Backend->SetBreakpoint(exception_triage, [](Backend_t *Backend) {

…
        const uint64_t rip =  g_Backend->VirtRead8(saved_rip);
        const uint64_t fault_address = g_Backend->VirtRead8(rsi+Gva_t(8));
        const std::string Filename = fmt::format(
            "crash-{:#x}-{:#x}-{:#x}", rdi, rip, fault_address);
        DebugPrint("Crash: {}\n", Filename);
        Backend->Stop(Crash_t(Filename));

      })) {
    return false;
  }
```

End condition

```
Gva_t retq = Gva_t(0xffffff800078ce28);
if (!g_Backend->SetBreakpoint(retq, [](Backend_t *Backend) {
      Backend->Stop(Ok_t());
    })) {
  return false;
}
```

cisco

On the fuzzer side of things, we need to prepare a harness.
First, we need to set up a couple of breakpoints that will let us catch crashes (note Backend->Stop(Crash)) And that would signal when testcase execution is complete.
Here we chose to end the testcase at the end (ret instruciton) of ip6_input function

Fuzzer harness cont.

Insert testcase

```
bool InsertTestcase(const uint8_t *Buffer, const size_t BufferSize) {

if (BufferSize < 40) return true; // mutated data too short

  Gva_t ipv6_header = Gva_t(0xffffff904e51b0d8);
  if(!g_Backend->VirtWriteDirty(ipv6_header,Buffer,40)){
    DebugPrint("VirtWriteDirtys failed\n");
  }

  Gva_t icmp6_data = Gva_t(0xffffff904e373000);
  if(!g_Backend->VirtWriteDirty(icmp6_data,Buffer+40,BufferSize-40)){
    DebugPrint("VirtWriteDirtys failed\n");
  }

  return true;
}
```

Second part of. The fuzzing harness is inserting the testcase into the memory of the fuzzed snapshot.
We already know the address of our mbuf structure from the debugger, so we simply write the mutated testcase there and make sure the structure is sane…

With that, the fuzzing can proceed.

It’s working

Sample fuzzing log

```
Running server on tcp://localhost:31337..
#0 cov: 0 (+0) corp: 0 (0.0b) exec/s: -nan (1 nodes) lastcov: 8.0s crash: 0 timeout: 0 cr3: 0 uptime: 8.0s
Saving output in .\outputs\4b20f7c59a0c1a03d41fc5c3c436db7c
Saving output in .\outputs\c6cc17a6c6d8fea0b1323d5acd49377c
Saving output in .\outputs\525101cf9ce45d15bbaaa8e05c6b80cd
Saving output in .\outputs\26c094dded3cf21cf241e59f5aa42a42
Saving output in .\outputs\b454d6965f113a025562ac9874446b7a
Saving output in .\outputs\00680b75d90e502fd0413c172aeca256
Saving output in .\outputs\51e31306ef681a8db35c74ac845bef7e
Saving output in .\outputs\b996cc78a4d3f417dae24b33d197defc
Saving output in .\outputs\2f456c73b5cd21fbaf647271e9439572
#10699 cov: 9778 (+9778) corp: 15 (9.1kb) exec/s: 1.1k (1 nodes) lastcov: 0.0s crash: 0 timeout: 0 cr3: 0 uptime: 18.0s
Saving output in .\outputs\3b93493ff98cf5e46c23a8b337d8242e
Saving output in .\outputs\73100aa4ae076a4cf29469ca70a360d9
#20922 cov: 9781 (+3) corp: 17 (10.0kb) exec/s: 1.0k (1 nodes) lastcov: 3.0s crash: 0 timeout: 0 cr3: 0 uptime: 28.0s
#31663 cov: 9781 (+0) corp: 17 (10.0kb) exec/s: 1.1k (1 nodes) lastcov: 13.0s crash: 0 timeout: 0 cr3: 0 uptime: 38.0s
#42872 cov: 9781 (+0) corp: 17 (10.0kb) exec/s: 1.1k (1 nodes) lastcov: 23.0s crash: 0 timeout: 0 cr3: 0 uptime: 48.0s
#53925 cov: 9781 (+0) corp: 17 (10.0kb) exec/s: 1.1k (1 nodes) lastcov: 33.0s crash: 0 timeout: 0 cr3: 0 uptime: 58.0s
#65054 cov: 9781 (+0) corp: 17 (10.0kb) exec/s: 1.1k (1 nodes) lastcov: 43.0s crash: 0 timeout: 0 cr3: 0 uptime: 1.1min
#75682 cov: 9781 (+0) corp: 17 (10.0kb) exec/s: 1.1k (1 nodes) lastcov: 53.0s crash: 0 timeout: 0 cr3: 0 uptime: 1.3min
Saving output in .\outputs\00f15aa5c6a1c822b36e33afb362e9ec
```

Finally , we can see the fuzzing in action. Here we see acouple of interesting things in the output.
Lines that begin with “saving output” mean that the fuzzer has uncovered new code , the code coverage is increasing which is good.
We can see a little bit about the stats , too. We are getting aobut a thousand executions per second with just a single node and the coverage is constantly increasing.
No crashes yet , but that’s ok .

Just to reiterate: we have captured a snapshot of a running macos VM , transplanted it onto whatever other platform (windows in this case) and started fuzzing it there. At this point, we can distribute this across hundreds of cores and utilize all the resources we have available on our regular off-the-shelf servers.

Additional support tooling

No fuzzing framework is complete without these

Testcase Tracing

Tracing testcase execution is essential for debugging the fuzzing harness

Symbolizing

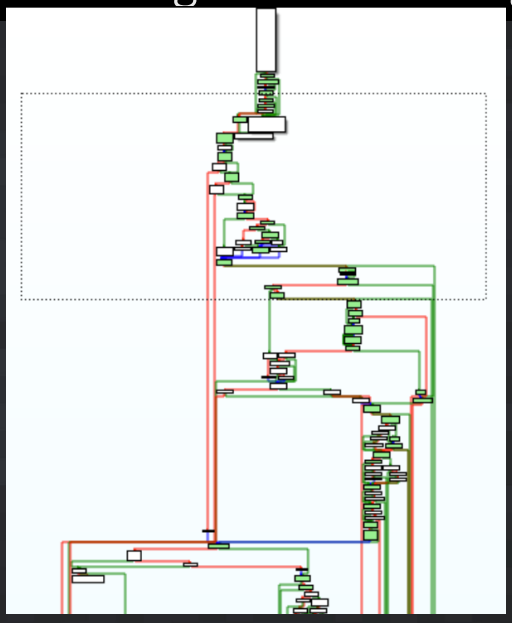
Kernel caches on macOS complicate things
Custom built kernel , symbols extraction through debugger
Symbolicated traces

Coverage analysis

Blindly fuzzing easily gets stuck
Using symbolized traces to manualy examine coverage
IDA Lighthouse

Tracing, symbolizing and vizualizing

```
ip6_input+0x36
ip6_input+0x40
ip6_input+0x4a
ip6_input+0x51
ip6_input+0x56
bzero
bzero+0x3
bzero+0x5
bzero+0x6
bzero+0x8
ip6_input+0x5b
ip6_input+0x66
ip6_input+0x10b
ip6_input+0x127
ip6_input+0x129
ip6_input+0x12e
ip6_input+0x130
m_tag_locate
m_tag_locate+0x1
m_tag_locate+0x4
m_tag_locate+0x8
m_tag_locate+0xa
m_tag_locate+0x37
m_tag_locate+0x4b
m_tag_locate+0x4d
m_tag_locate+0x4e
ip6_input+0x135
ip6_input+0x138
ip6_input+0x145
```



cisco | Talos

macOS KASAN

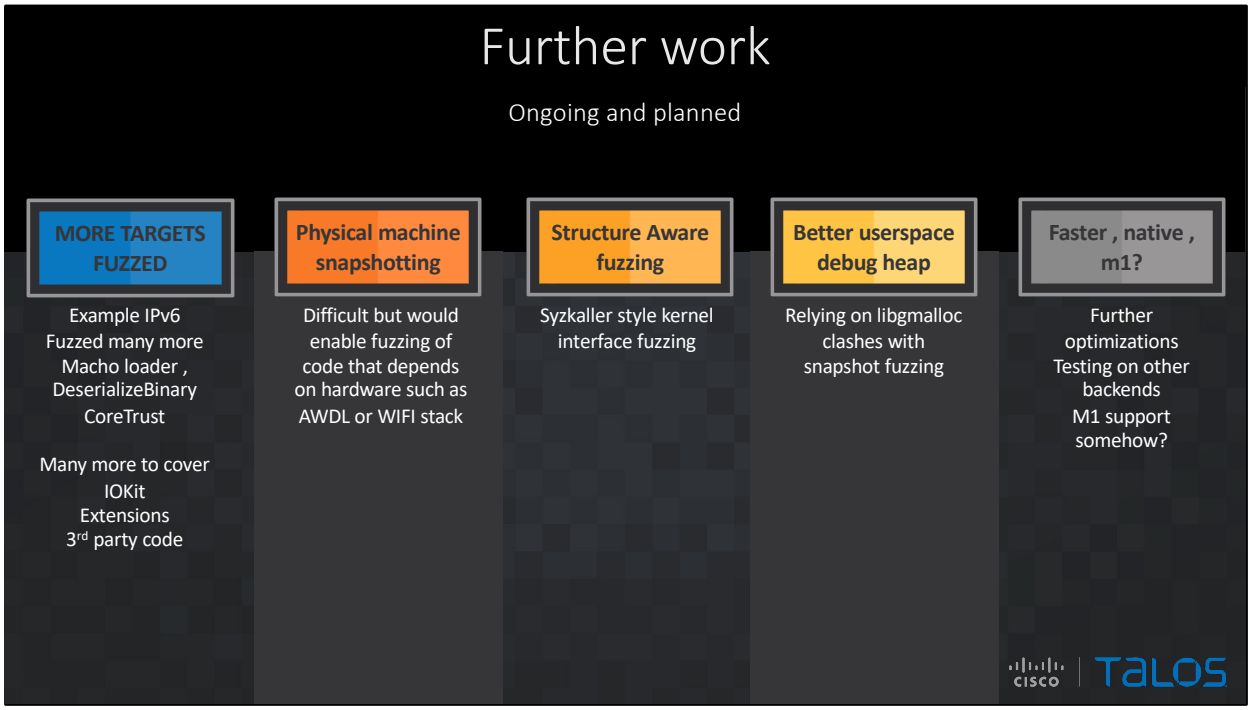
Essential for effective fuzzing

```
  frame #14: 0xffffff801197da1c kernel.kasan`handle_debugger_trap [inlined] debugger_collect_diagnostics(exception=6, code=3, subcode=0, state=0xffffffa0927ef8e0) at debug.c:130
    frame #15: 0xffffff801197d657 kernel.kasan`handle_debugger_trap(exception=6, code=3, subcode=0, state=0xffffffa0927ef8e0) at debug.c:1423:3 [opt]
    frame #16: 0xffffff8011d79544 kernel.kasan`kdp_i386_trap(trapno=<unavailable>, saved_state=0xffffffa0927ef8e0, result=<unavailable>, va=0x0000000109e4d020) at kdp_machdep.c
    frame #17: 0xffffff8011d62f1a kernel.kasan`kernel_trap(state=0xffffffa0927ef290, lo_spp=<unavailable>) at trap.c:772:7 [opt]
    frame #18: 0xffffff8011d81af0 kernel.kasan`trap_from_kernel + 38
    frame #19: 0xffffff801197d0f0 kernel.kasan`DebuggerTrapWithState [inlined] current_debugger_state at debug.c:182:9 [opt]
    frame #20: 0xffffff801197d0f0 kernel.kasan`DebuggerTrapWithState(db_op=DBOP_PANIC, db_message="panic", db_panic_str="Kernel trap at 0x%016llx, type %d=%s, registers:\nCR0: 
0x%016llx\nR8:  0x%016llx, R9:  0x%016llx, R10: 0x%016llx, R11: 0x%016llx\nR12: 0x%016llx, R13: 0x%016llx, R14: 0x%016llx, R15: 0x%016llx\nRFL: 0x%016llx, RIP: 0x%016llx, CS:  
db_panic_data_ptr=0x0000000000000000, db_proceed_on_sync_failure=1, db_panic_caller=18446743524252992714) at debug.c:667:8 [opt]
    frame #21: 0xffffff801197dec0 kernel.kasan`panic_trap_to_debugger(panic_format_str="Kernel trap at 0x%016llx, type %d=%s, registers:\nCR0: 0x%016llx, CR2: 0x%016llx, CR3: 0
R10: 0x%016llx, R11: 0x%016llx\nR12: 0x%016llx, R13: 0x%016llx, R14: 0x%016llx, R15: 0x%016llx\nRFL: 0x%016llx, RIP: 0x%016llx, CS:  0x%016llx, SS:  0x%016llx\nFault CR2: 0x%01
panic_data_ptr=0x0000000000000000, panic_caller=18446743524252992714) at debug.c:1060:2 [opt]
    frame #22: 0xffffff80132427b5 kernel.kasan`panic(str=<unavailable>) at debug.c:877:2 [opt]
    frame #23: 0xffffff8011d644ca kernel.kasan`panic_trap(regs=<unavailable>, pl=<unavailable>, fault_result=<unavailable>) at trap.c:839:2 [opt]
    frame #24: 0xffffff8011d632a9 kernel.kasan`kernel_trap(state=0xffffffa0927ef8d0, lo_spp=<unavailable>) at trap.c:778:2 [opt]
    frame #25: 0xffffff8011d81af0 kernel.kasan`trap_from_kernel + 38
    frame #26: 0xffffff80132351fe kernel.kasan`test_heap_uaf(t=<unavailable>) at kasan-test.c:134:7 [opt]
    frame #27: 0xffffff8013234b67 kernel.kasan`kasan_run_test(test_list=0xffffff8014005000, testno=4, fail=0) at kasan-test.c:596:13 [opt]
    frame #28: 0xffffff8013234dea kernel.kasan`sysctl_kasan_test [inlined] kasan_test(testno=1, fail=0) at kasan-test.c:637:10 [opt]
    frame #29: 0xffffff8013234da3 kernel.kasan`sysctl_kasan_test(oidp=<unavailable>, arg1=<unavailable>, arg2=<unavailable>, req=<unavailable>) at kasan-test.c:677:3 [opt]
    frame #30: 0xffffff80129e3cc5 kernel.kasan`sysctl_root(from_kernel=<unavailable>, string_is_canonical=<unavailable>, namestring=<unavailable>, namestringlen=<unavailable>, 
    frame #31: 0xffffff80129e4556 kernel.kasan`sysctl [inlined] userland_sysctl(string_is_canonical=0, namestring="kern.kasan.test", namestringlen=1024, name=0xffffffa0927efd30
    frame #32: 0xffffff80129e44f2 kernel.kasan`sysctl(p=<unavailable>, uap=<unavailable>, retval=<unavailable>) at kern_newsysctl.c:1928:10 [opt]
    frame #33: 0xffffff8012de6f42 kernel.kasan`unix_syscall64(state=<unavailable>) at systemcalls.c:394:10 [opt]
    frame #34: 0xffffff8011d822b6 kernel.kasan`hndl_unix_scall64 + 22
```

one other thing that's essential when fuzzing is making sure memory corruptions get detected.

XNU is open source for the most part and apple is nice enough to provide us with KDK that contains a kernel build with address sanitizer enabled.

This is essential for fuzzing , otherwise subtle bugs would easily go uncaught.
You need to jump through a lot of hoops to get it enabled. But in the end it does work. This is how a kernel crash looks like in address sanitizer. We'd definitely be able to catch this easily in our fuzzing harness.



There's further work to be done with this project.

Due to time, i've only shown one example fuzzing target, but we've covered a lot more. Some obvious, some less so and there's still plenty to target.

Next on my todo list is figuring out physical machine snapshotting which would enable us to fuzz some hardware dependent components. Like AWDL or Wifi stack in general.

Additionally, performing syzkaller style fuzzing of certain targets would be very interesting.

And finally, in the realm of wishful thinking for now, it would be very cool if we had a way to achieve the same capabilities for testing apple silicon code.

And with that, i would love to hear if you have any questions.
If we run out of time here, i'll be around the conference , of course.

Thank you!


cisco
TALOS
TALOSINTELLIGENCE.COM