Fuzzing on Windows

Leo Adrien

whoami

- Former pentester
- Currently:
  - Postgrad computer science at Monash
  - Independent security researcher focussing on Windows
- Been doing ~hacking for about 5 years now

  australeo {@infosec.exchange | twitter | github}

Outline

Intro to fuzzing:

- What is it?
- What are the components of a fuzzer?

How to fuzz binary-only (closed-source) programs on Windows:

- Case study (CVE-2022-21884)
  - Background
  - Creating a persistent fuzzing harness
- Summary / Discussion

Fuzzing

"... the systematic testing of programs via the generation of novel or unexpected inputs." [1]

[1] https://dl.acm.org/doi/pdf/10.1145/3588045

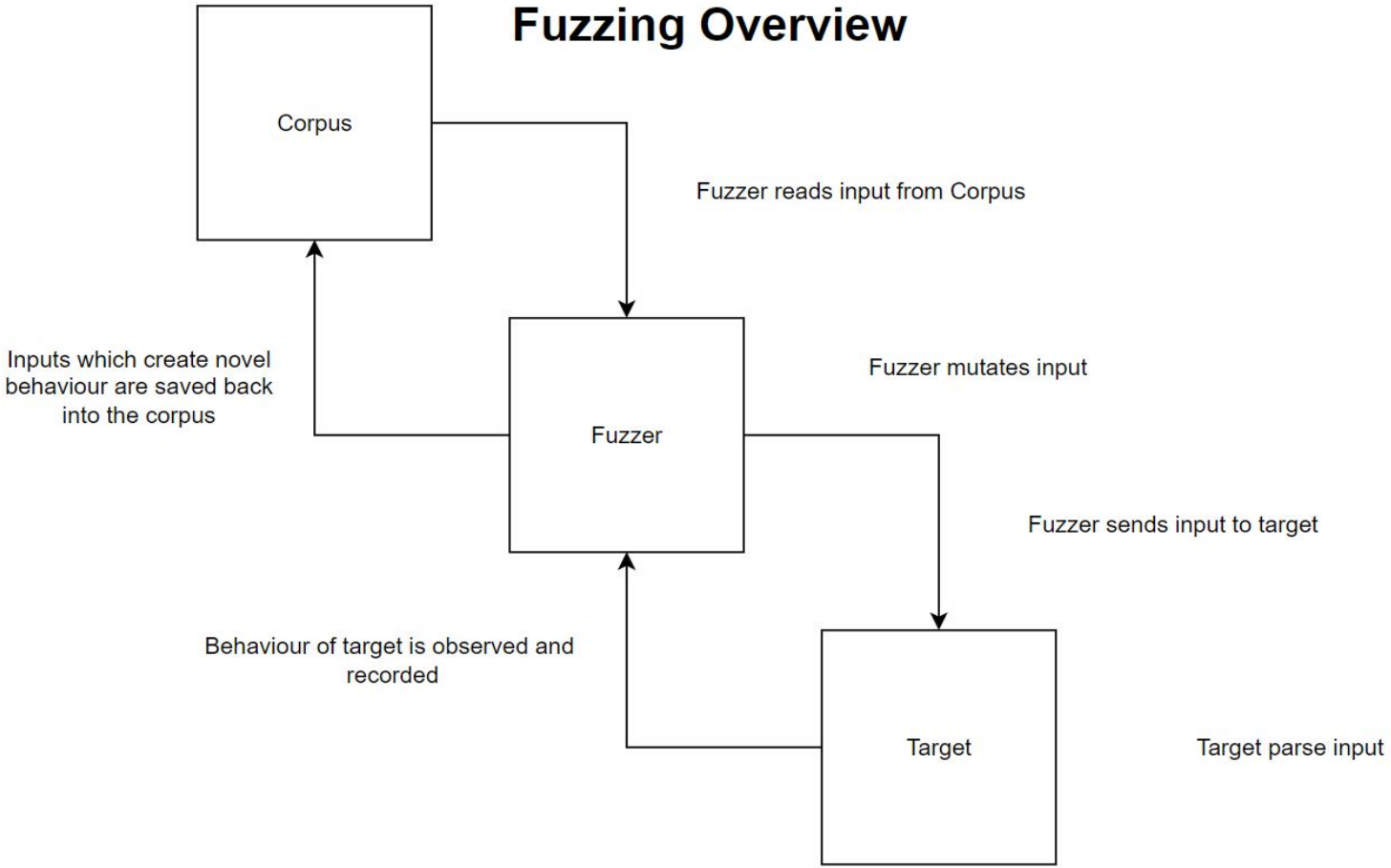
Fuzzing

```
cat /dev/urandom | target
```

Fuzzing

Demo time!

Fuzzing Overview



Fuzzer - Corpus

Collection of interesting inputs which conform to the target's expected format. Mutating existing samples will always be more effective than starting with random data.

Most fuzzers will add to this collection automatically by saving *interesting* mutations back to the corpus (more on this in **Coverage**).

Fuzzer - Mutator

A function which takes an input from the corpus and performs some kind of mutation/malformation.

Can be as simple as flipping a random bit of the input, but more complex mutations are commonly used.

Most fuzzers will have a range of mutations available and choose one each time a new test case is generated.

Fuzzer - Harness

A program or function which calls the target function and provides it the input.

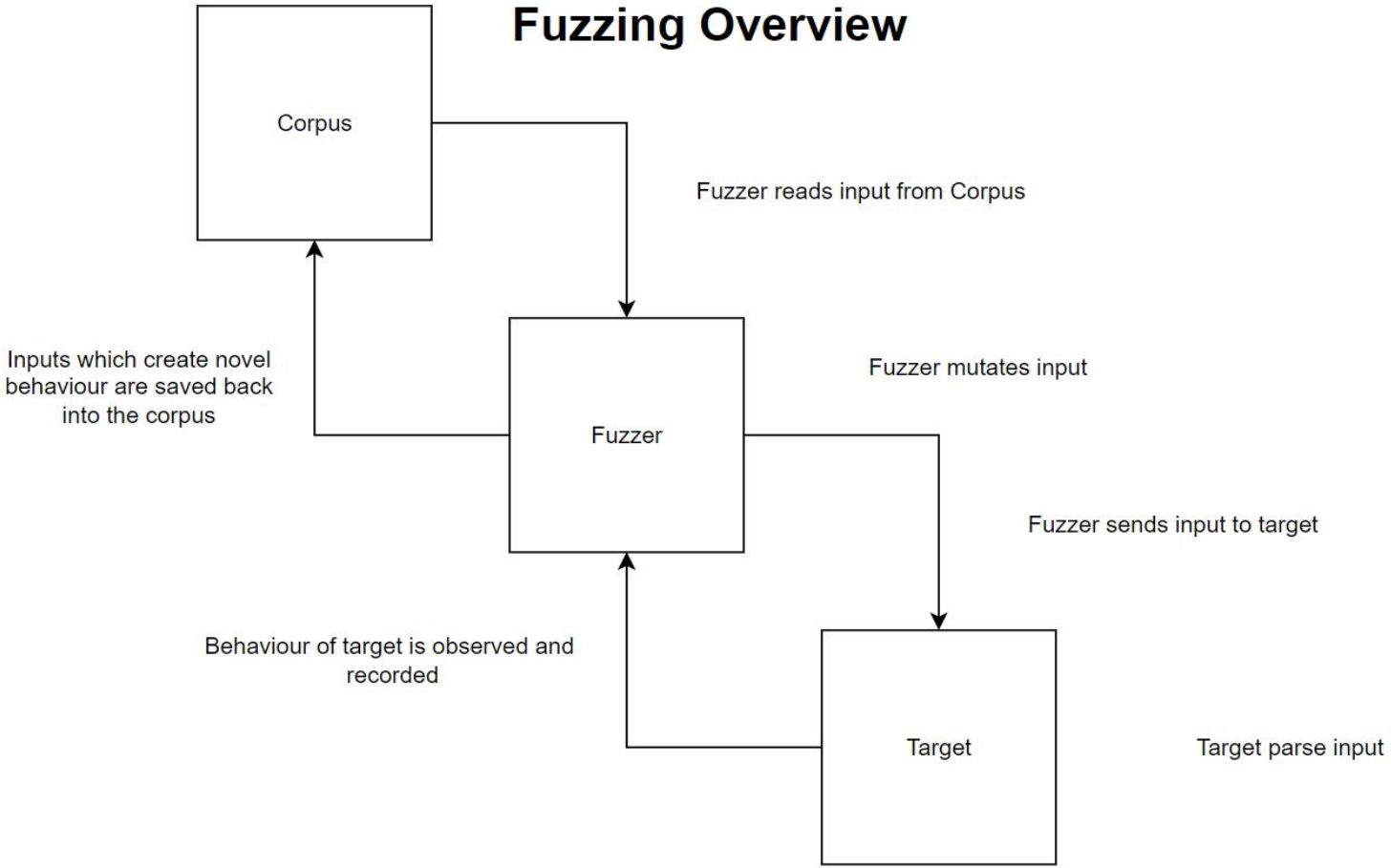
This may involve creating a new process and providing the path to a mutated file, or may occur entirely *in-process* and consist of calling a function and passing a pointer to a mutated byte array.

Fuzzer - Observer

A program which observes the harness and monitors for crashes, or some other oracle value which detects that an unhandled state has occurred.

May also collect coverage information, such as the functions called, basic blocks executed (block coverage), or basic block edges traversed (edge coverage).

Fuzzing Overview



Types of Fuzzer

Can be broken down into a few categories:

By execution method:

- unmodified, persistent, snapshot

By input generation:

- random, mutational, generational

By coverage:

- blackbox, block, edge, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- **unmodified**, persistent, snapshot

By input generation:

- random, mutational, generational

By coverage:

- blackbox, block, edge, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- in-process, **persistent**, snapshot

By input generation:

- random, mutational, generational

By coverage:

- blackbox, block, edge, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- in-process, persistent, **snapshot**

By input generation:

- random, mutational, generational

By coverage:

- blackbox, block, edge, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- in-process, persistent, snapshot

By input generation:

- **random**, mutational, generational

By coverage:

- blackbox, block, edge, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- in-process, persistent, snapshot

By input generation:

- random, **mutational**, generational

By coverage:

- blackbox, block, edge, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- in-process, persistent, snapshot

By input generation:

- random, mutational, **generational**

By coverage:

- blackbox, block, edge, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- in-process, persistent, snapshot

By input generation:

- random, mutational, generational

By coverage:

- **blackbox**, block, edge, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- in-process, persistent, snapshot

By input generation:

- random, mutational, generational

By coverage:

- blackbox, **block, edge**, oracle

Types of fuzzer

Can be broken down into a few categories:

By execution method:

- in-process, persistent, snapshot

By input generation:

- random, mutational, generational

By coverage:

- blackbox, block, edge, **oracle**

Fuzzing on Windows

Persistent/in-process Fuzzers:

- Jackalope, WinAFL, LibAFL-Frida

Snapshot Fuzzers:

- WTF (What The Fuzz)

Case study: CVE-2022-21884

Background:

- Out-of-Bounds Write (heap buffer overflow) in Windows' Local Security Authority Subsystem Service (LSASS)
- Discovered with fuzzing using Jackalope in 2021, reported/fixed belatedly in 2022

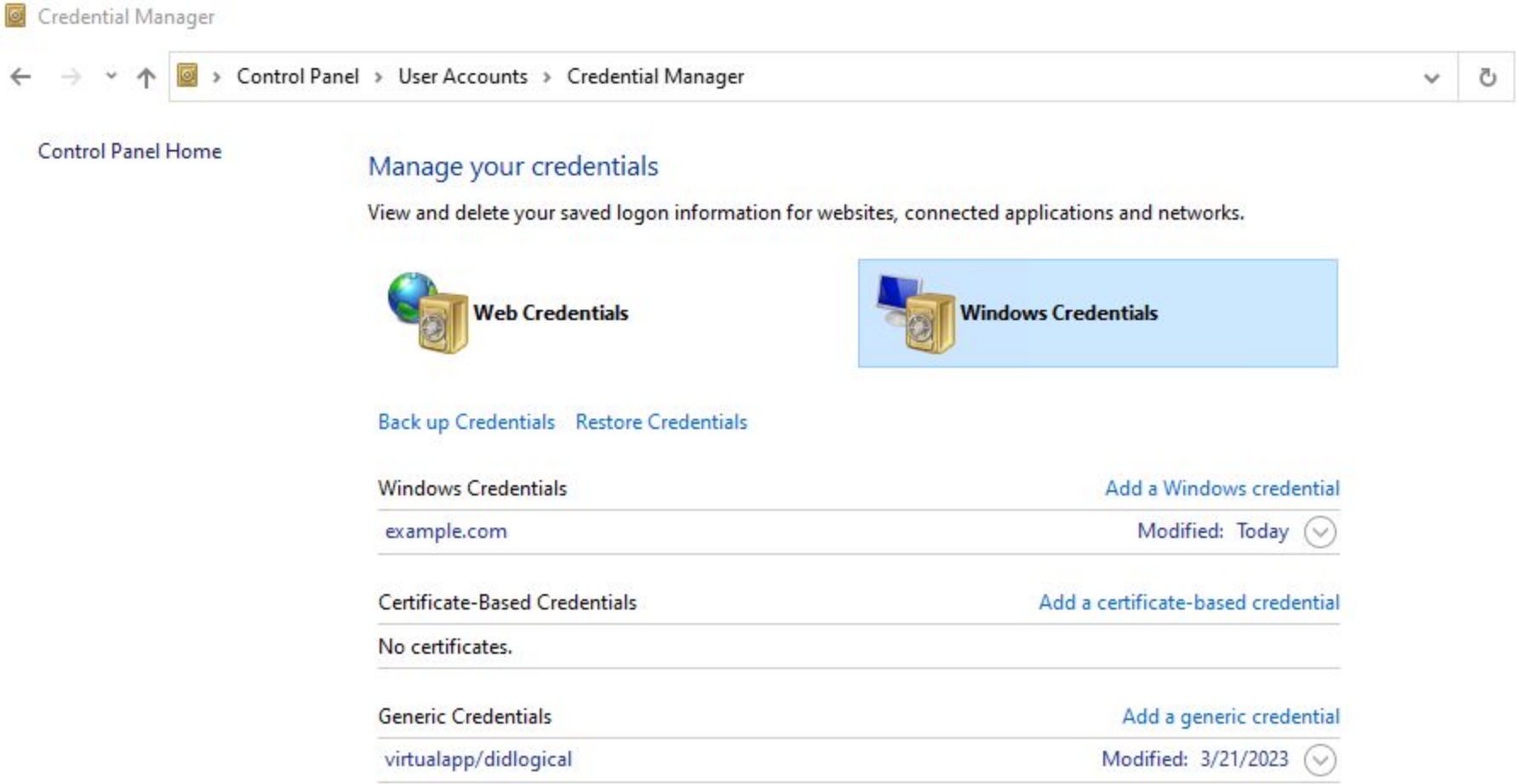
Case study: CVE-2022-21884

Background:

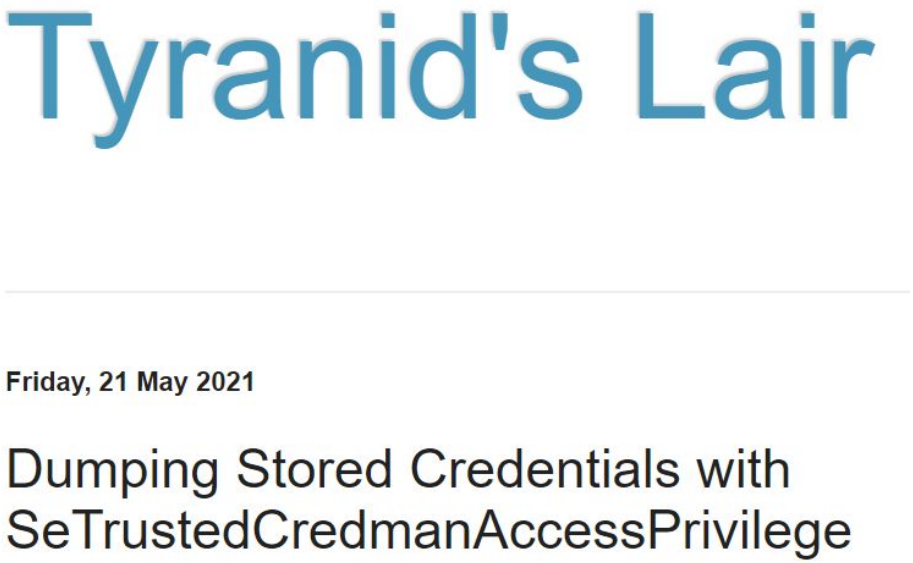
- LSASS manages local authentication for Windows
  - Login/logout, credential storage
  - Runs as Windows service
  - Provides API via Remote Procedure Call (RPC) interface
  - Primary target of post-exploitation tools like *mimikatz*

Case study: CVE-2022-21884

- Background:
  - LSASS also provides miscellaneous credential related functions, such as storing credentials for other applications/systems in a “Credential Manager”
  - Not commonly used, especially in the age of password 3rd-party managers


Credential Manager
Control Panel › User Accounts › Credential Manager
Control Panel Home
Manage your credentials
View and delete your saved logon information for websites, connected applications and networks.
Web Credentials
Windows Credentials
Back up Credentials
Restore Credentials
Windows Credentials
Add a Windows credential
example.com
Modified: Today
Certificate-Based Credentials
Add a certificate-based credential
No certificates.
Generic Credentials
Add a generic credential
virtualapp/didlogical
Modified: 3/21/2023

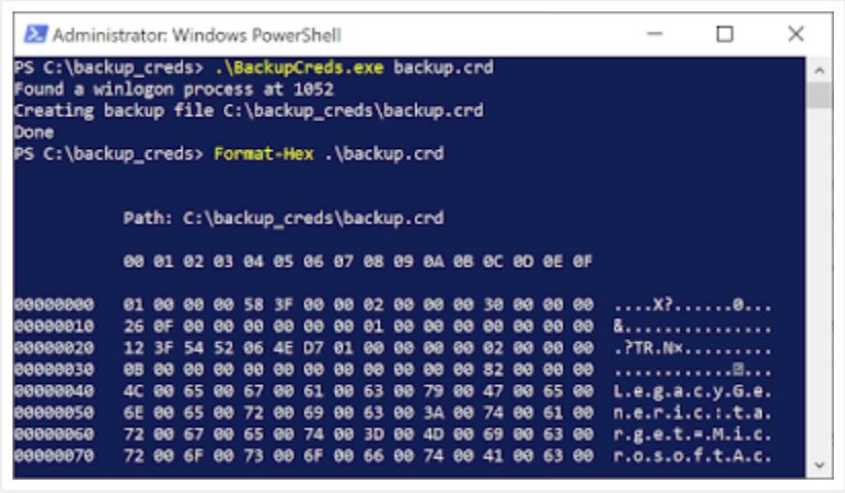
Case study: CVE-2022-21884



https://www.tiraniddo.dev/2021/05/dumping-stored-credentials-with.html

Case study: CVE-2022-21884

If it all goes well you'll have all the of the user's credentials in a packed binary format. I couldn't immediately find anyone documenting it, but people obviously have done before. I'll leave doing all this yourself as a exercise for the reader. I don't feel like providing an implementation.



https://www.tiraniddo.dev/2021/05/dumping-stored-credentials-with.html

Case study: CVE-2022-21884

Step 1:

- Reconnaissance (Where does the parsing function exist?)
  - Forshaw's blog tells us there is a function called *CredrBackupCredentials* so start by looking for that.
  - End up in *lsasrv.dll*, throw it into RE tool of choice
  - We actually are interested in the unpacking of the credentials, so end up with *CredrRestoreCredentials*
  - Eventually find *CredpUnmarshalCredential*

Case study: CVE-2022-21884

Demo time!

Case study: CVE-2022-21884

```
0: kd> x lsasrv!CredpUnmarshalCredential
00007ffb`234ef164 lsasrv!CredpUnmarshalCredential (void)
0: kd> bp lsasrv!CredpUnmarshalCredential
0: kd> g
Breakpoint 0 hit
lsasrv!CredpUnmarshalCredential:
0033:00007ffb`234ef164 4055            push    rbp
3: kd> k
 # Child-SP          RetAddr               Call Site
00 0000001f`eacfea18 00007ffb`235537a6     lsasrv!CredpUnmarshalCredential
01 0000001f`eacfea20 00007ffb`2358672b     lsasrv!CrediRestoreCredentials+0x882
02 0000001f`eacfeb40 00007ffb`260949d3     lsasrv!CredrRestoreCredentials+0x8b
03 0000001f`eacfeb90 00007ffb`260fa8f9     RPCRT4!Invoke+0x73
04 0000001f`eacfec00 00007ffb`2607b1ac     RPCRT4!Ndr64StubWorker+0xb79
```

Case study: CVE-2022-21884

Demo time!

Case study: CVE-2022-21884

```
long CredpUnmarshalCredential(uint8_t* __ptr64 arg1,
    unsigned long arg2, unsigned long* __ptr64 arg3,
    struct _CANONICAL_CREDENTIAL* __ptr64* __ptr64 arg4)
```

Notes as a (fairly experienced) reverse engineer of Windows binaries:

- *char*/u8** followed by a u32?

  -> BYTE pcBuffer, ULONG cbBuffer*

- *struct SomeStruct ** ?*

  *-> struct SomeStruct** ppOutputStruct*

```
2: kd> g
Breakpoint 0 hit
lsasrv!CredpUnmarshalCredential:
0033:00007ffb`234ef164 4055            push    rbp
3: kd> r rcx; r rdx; r r8; r r9
rcx=000001fc3bfa6620
rdx=0000000000002c6c
r8=0000001feaa7e680
r9=0000001feaa7e650
3: kd> !heap -x @rcx
[50 Percent Complete]
Search was performed for the following Address: 0x000001fc3bfa6620
Below is a detailed information about this address.
Heap Address              : 0x000001fc3b8e0000
The address was found in backend heap.
Segment Address           : 0x000001fc3bf00000
Segment Unit size         : 4096 Bytes
Page range index (0-255)  : 162
Page descriptor address   : 0x000001fc3bf01440
Subsegment address        : 0x000001fc3bfa2000
Subsegment Size           : 11392 Bytes
Allocation status         : Variable size allocated chunk.
Chunk header address      : 000001fc3bfa6610
Chunk size (bytes)        : 11392
Chunk unused bytes        : 0

3: kd> ? 0n11392
Evaluate expression: 11392 = 00000000`00002c80
```

```
2: kd> g
Breakpoint 0 hit
lsasrv!CredpUnmarshalCredential:
0033:00007ffb`234ef164 4055            push    rbp
3: kd> r rcx; r rdx; r r8; r r9
rcx=000001fc3bfa6620
rdx=0000000000002c6c
r8=0000001feaa7e680
r9=0000001feaa7e650
3: kd> !heap -x @rcx
[50 Percent Complete]
Search was performed for the following Address: 0x000001fc3bfa6620
Below is a detailed information about this address.
Heap Address              : 0x000001fc3b8e0000
The address was found in backend heap.
Segment Address           : 0x000001fc3bf00000
Segment Unit size         : 4096 Bytes
Page range index (0-255)  : 162
Page descriptor address   : 0x000001fc3bf01440
Subsegment address        : 0x000001fc3bfa2000
Subsegment Size           : 11392 Bytes
Allocation status         : Variable size allocated chunk.
Chunk header address      : 000001fc3bfa6610
Chunk size (bytes)        : 11392
Chunk unused bytes        : 0

3: kd> ? 0n11392
Evaluate expression: 11392 = 00000000`00002c80
```

```
2: kd> g
Breakpoint 0 hit
lsasrv!CredpUnmarshalCredential:
0033`00007ffb`234ef164 4055            push    rbp
3: kd> r rcx; r rdx; r r8; r r9
rcx=000001fc3bfa6620
rdx=0000000000002c6c
r8=0000001feaa7e680
r9=0000001feaa7e650
3: kd> !heap -x @rcx
[50 Percent Complete]
Search was performed for the following Address: 0x000001fc3bfa6620
Below is a detailed information about this address.
Heap Address              : 0x000001fc3b8e0000
The address was found in backend heap.
Segment Address           : 0x000001fc3bf00000
Segment Unit size         : 4096 Bytes
Page range index (0-255)  : 162
Page descriptor address   : 0x000001fc3bf01440
Subsegment address        : 0x000001fc3bfa2000
Subsegment Size           : 11392 Bytes
Allocation status         : Variable size allocated chunk.
Chunk header address      : 000001fc3bfa6610
Chunk size (bytes)        : 11392
Chunk unused bytes        : 0

3: kd> ? 0n11392
Evaluate expression: 11392 = 00000000`00002c80
```

```
2: kd> g
Breakpoint 0 hit
lsasrv!CredpUnmarshalCredential:
0033`00007ffb`234ef164 4055            push    rbp
3: kd> r rcx; r rdx; r r8; r r9
rcx=000001fc3bfa6620
rdx=0000000000002c6c
r8=0000001feaa7e680
r9=0000001feaa7e650
3: kd> !heap -x @rcx
[50 Percent Complete]
```

```
long CredpUnmarshalCredential(uint8_t* __ptr64 arg1,
    unsigned long arg2, unsigned long* __ptr64 arg3,
    struct _CANONICAL_CREDENTIAL* __ptr64* __ptr64 arg4)
```

RCX

RDX

R8

R9

```
Segment Unit Size          : 4096 Bytes
Page range index (0-255)   : 162
Page descriptor address    : 0x000001fc3bf01440
Subsegment address         : 0x000001fc3bfa2000
Subsegment Size            : 11392 Bytes
Allocation status          : Variable size allocated chunk.
Chunk header address       : 000001fc3bfa6610
Chunk size (bytes)         : 11392
Chunk unused bytes         : 0

3: kd> ? 0n11392
Evaluate expression: 11392 = 00000000`00002c80
```

```
2: kd> g
Breakpoint 0 hit
lsasrv!CredpUnmarshalCredential:
0033:00007ffb`234ef164 4055            push    rbp
3: kd> r rcx; r rdx; r r8; r r9
rcx=000001fc3bfa6620
rdx=0000000000002c6c
r8=0000001feaa7e680
r9=0000001feaa7e650
3: kd> !heap -x @rcx
[50 Percent Complete]
Search was performed for the following Address: 0x000001fc3bfa6620
Below is a detailed information about this address.
Heap Address              : 0x000001fc3b8e0000
The address was found in backend heap.
Segment Address           : 0x000001fc3bf00000
Segment Unit size         : 4096 Bytes
Page range index (0-255)  : 162
Page descriptor address   : 0x000001fc3bf01440
Subsegment address        : 0x000001fc3bfa2000
Subsegment Size           : 11392 Bytes
Allocation status         : Variable size allocated chunk.
Chunk header address      : 000001fc3bfa6610
Chunk size (bytes)        : 11392
Chunk unused bytes        : 0

3: kd> ? 0n11392
Evaluate expression: 11392 = 00000000`00002c80
```

```
2: kd> g
Breakpoint 0 hit
lsasrv!CredpUnmarshalCredential:
0033:00007ffb`234ef164 4055            push    rbp
3: kd> r rcx; r rdx; r r8; r r9
rcx=000001fc3bfa6620
rdx=0000000000002c6c
r8=0000001feaa7e680
r9=0000001feaa7e650
3: kd> !heap -x @rcx
[50 Percent Complete]
Search was performed for the following Address: 0x000001fc3bfa6620
Below is a detailed information about this address.
Heap Address              : 0x000001fc3b8e0000
The address was found in backend heap.
Segment Address           : 0x000001fc3bf00000
Segment Unit size         : 4096 Bytes
Page range index (0-255)  : 162
Page descriptor address   : 0x000001fc3bf01440
Subsegment address        : 0x000001fc3bfa2000
Subsegment Size           : 11392 Bytes
Allocation status         : Variable size allocated chunk.
Chunk header address      : 000001fc3bfa6610
Chunk size (bytes)        : 11392
Chunk unused bytes        : 0

3: kd> ? 0n11392
Evaluate expression: 11392 = 00000000`00002c80
```

```
2: kd> g
Breakpoint 0 hit
lsasrv!CredpUnmarshalCredential:
0033`00007ffb`234ef164 4055            push    rbp
3: kd> r rcx; r rdx; r r8; r r9
rcx=000001fc3bfa6620
rdx=0000000000002c6c
r8=0000001feaa7e680
r9=0000001feaa7e650
3: kd> !heap -x @rcx
[50 Percent Complete]
```

```
long CredpUnmarshalCredential(uint8_t* __ptr64 arg1,
    unsigned long arg2, unsigned long* __ptr64 arg3,
    struct _CANONICAL_CREDENTIAL* __ptr64* __ptr64 arg4)
```

RCX

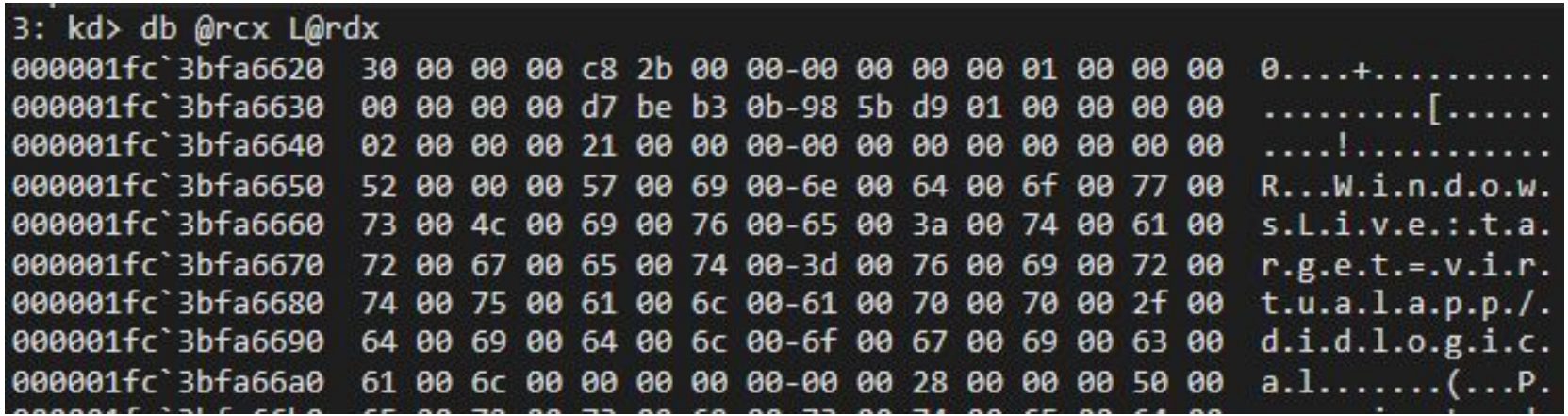
RDX

R8

R9

```
Segment Unit Size          : 4096 Bytes
Page range index (0-255)   : 162
Page descriptor address    : 0x000001fc3bf01440
Subsegment address         : 0x000001fc3bfa2000
Subsegment Size            : 11392 Bytes
Allocation status          : Variable size allocated chunk.
Chunk header address       : 000001fc3bfa6610
Chunk size (bytes)         : 11392
Chunk unused bytes         : 0

3: kd> ? 0n11392
Evaluate expression: 11392 = 00000000`00002c80
```


3: kd> db @rcx L@rdx
000001fc`3bfa6620 30 00 00 00 c8 2b 00 00-00 00 00 00 01 00 00 00 0....+..........
000001fc`3bfa6630 00 00 00 00 d7 be b3 0b-98 5b d9 01 00 00 00 00 ....[......
000001fc`3bfa6640 02 00 00 00 21 00 00 00-00 00 00 00 00 00 00 00 ....!...........
000001fc`3bfa6650 52 00 00 00 57 00 69 00-6e 00 64 00 6f 00 77 00 R...W.i.n.d.o.w.
000001fc`3bfa6660 73 00 4c 00 69 00 76 00-65 00 3a 00 74 00 61 00 s.L.i.v.e.:.t.a.
000001fc`3bfa6670 72 00 67 00 65 00 74 00-3d 00 76 00 69 00 72 00 r.g.e.t.=.v.i.r.
000001fc`3bfa6680 74 00 75 00 61 00 6c 00-61 00 70 00 70 00 2f 00 t.u.a.l.a.p.p./.
000001fc`3bfa6690 64 00 69 00 64 00 6c 00-6f 00 67 00 69 00 63 00 d.i.d.l.o.g.i.c.
000001fc`3bfa66a0 61 00 6c 00 00 00 00 00-00 00 28 00 00 00 50 00 a.l.......(...P.

Case study: CVE-2022-21884

```
long CredpUnmarshalCredential(u8* pPackedBuffer,
    u32 PackedBufferLength, u32* pUnknownValue32,
    void** ppCanonicalCredStructOut)
```

The final* prototype of our target

Case study: CVE-2022-21884

Step 1:

- Reconnaissance
    - Got a function address and prototype

Step 2:

- Harness the target
    - Persistent fuzzing
        - Create a “harness” program which can load the target DLL and call the target function with our mutated input

Case study: CVE-2022-21884

Jackalope

- Coverage-guided fuzzer released by Ivan Fratric at Google in 2020
- Uses a library called TinyInst to dynamically instrument a given binary, in order to collect code coverage.
- Written in (nice) C++, very easy to read and/or customize
- Can communicate with a target program via files or shared memory

Case study: CVE-2022-21884

Harnessing

- We want to get Jackalope to send fuzz input to our target function via shared memory and observe the outcome
- Our target function is inside the lsasrv DLL

Case study: CVE-2022-21884

Harnessing

- A dynamic link library (DLL) is a shared object file (like a *nix .so) which can be loaded at runtime by other programs.
- By design, you can only call functions in a DLL which are specifically exported for external use.

```
declspec(dllexport) void ExportedFunction(int parameter);
```

Case study: CVE-2022-21884

Harnessing

- To manually load a DLL on Windows you can use the *LoadLibrary* function which will return a handle* to the newly loaded binary.

- Provide this handle* to the *GetProcAddress* function along with the target function’s name, and it will look it up in the export table and provide a pointer to the exported function.

Case study: CVE-2022-21884

Harnessing

- To manually load a DLL on Windows you can use the *LoadLibrary* function which will return a pointer to the newly loaded binary's base address.

- You can then manually add the offset to any function (or instruction) in the binary, and cast it to a function pointer of the desired type.

Harness walkthrough

Administrator: Windows PowerShell - fuzzer.exe -in C:\Fuzzing\in -out C:\Fuzzing\out -t 1000 -nthreads 2 -dump_coverage -delivery shmem -instrume...

```
Crashes: 24 (2 unique)
Hangs: 0
Offsets: 1258
Execs/s: 8545
Fuzzing sample 00033
Fuzzing sample 00046
Instrumented module lsasrv.dll, code size: 1171456
Fuzzing sample 00053
Fuzzing sample 00005
Fuzzing sample 00059
Fuzzing sample 00045
Fuzzing sample 00048
Fuzzing sample 00030

Total execs: 428928
Unique samples: 76 (0 discarded)
Crashes: 24 (2 unique)
Hangs: 0
Offsets: 1258
Execs/s: 8647
Fuzzing sample 00019
Fuzzing sample 00058
Instrumented module lsasrv.dll, code size: 1171456
Fuzzing sample 00066
Fuzzing sample 00025
Fuzzing sample 00049
Fuzzing sample 00028
Fuzzing sample 00051
Fuzzing sample 00038
```

Fuzzing on Windows

Persistent fuzzer pitfalls (Jackalope, WinAFL, LibAFL-Frida/tinyinst)

- Harnessing can be very difficult for “deep” targets
- State is the enemy
  - Can lead to instability, resource leaks, “fake” crashes
- Current tooling doesn’t work for kernel-mode targets

Case study: CVE-2022-21884

A quick note on sanitizers:

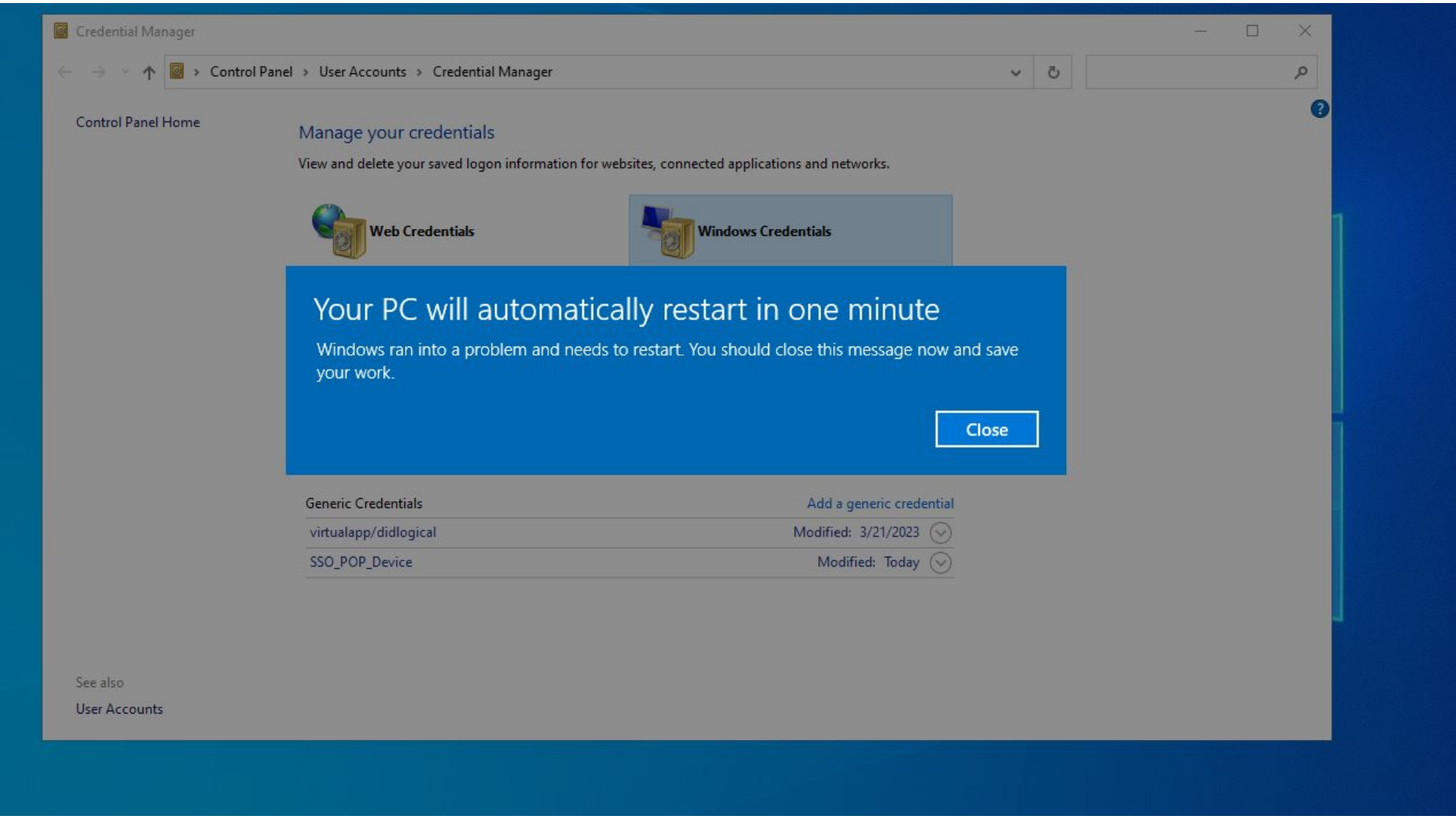
- When compiling code you can add sanitizers (ASAN, TSAN, UBSAN).
- For binary-only targets on Windows, enable runtime debug allocators:
  - For usermode: `gflags /i <program.exe> +hpa` (PageHeap)
  - For drivers: `verifier.exe`, select `Special Pool`, then the drivers you are targeting
- These make programs crash much more consistently!

Case study: CVE-2022-21884

Got a fuzzer running… now what?

Case study: CVE-2022-21884

- Inspect coverage
  - What paths are not being taken?
  - Why?
- Add strings to corpus to pass more checks


Credential Manager
Control Panel › User Accounts › Credential Manager
Control Panel Home
Manage your credentials
View and delete your saved logon information for websites, connected applications and networks.
Web Credentials
Windows Credentials
Your PC will automatically restart in one minute
Windows ran into a problem and needs to restart. You should close this message now and save your work.
Close
Generic Credentials
Add a generic credential
virtualapp/didlogical
Modified: 3/21/2023
SSO_POP_Device
Modified: Today
See also
User Accounts

Fuzzing on Windows: Summary

- Be on the lookout for fuzzable attack surface
- Windbg is your best friend
- Don’t forget the sanitizers (gflags, verifier)
- Coverage is for humans too!

Resources

Windows fuzzing:

https://github.com/googleprojectzero/Jackalope

https://github.com/0vercl0k/wtf

https://github.com/googleprojectzero/winafl

Linux/Open Source fuzzing:

https://github.com/AFLplusplus/AFLplusplus

https://github.com/AFLplusplus/LibAFL