CRAX plusplus

Modular Exploit Generator using Dynamic Symbolic Execution

@aesophor

HITCON PEACE 2022 | SURVIVAL GUIDE FOR THE CYBER WAR

Aug 19, HITCON PEACE 2022

Whoami

- aesophor
  - Software Engineer at Synology
  - MS degree: Software Quality Lab, NYCU
  - Talks:
    - HITCON 2022 - Today’s talk
    - SITCON 2019 - Writing an X11 tiling window manager

About SQLab

- Prof. Shih-Kun Huang (黃世昆)
- Current members:
  - Ph.D student * 2
  - MS student * 12
- Research:
  - Fuzzing
  - Exploit Generation

About SQLab

- Prof. Shih-Kun Huang (黃世昆)
- Current members:
  - Ph.D student * 2
  - MS student * 12
- Research:
  - Fuzzing
  - Exploit Generation

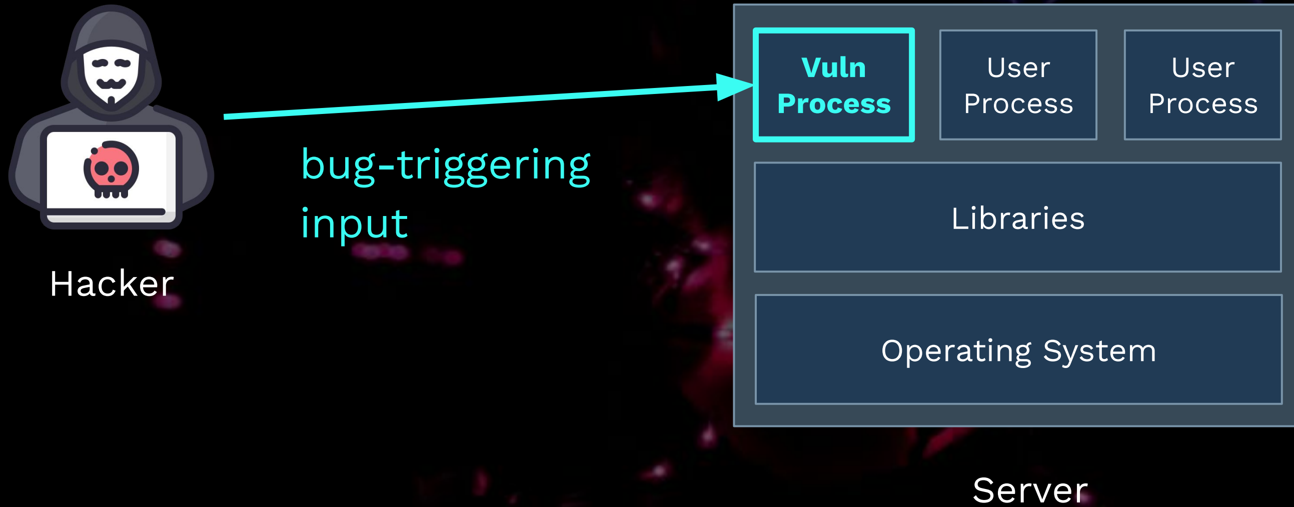
Disclaimer

CRAX++ uses the idea from other AEG research.
(Currently CRAX and LAEG)

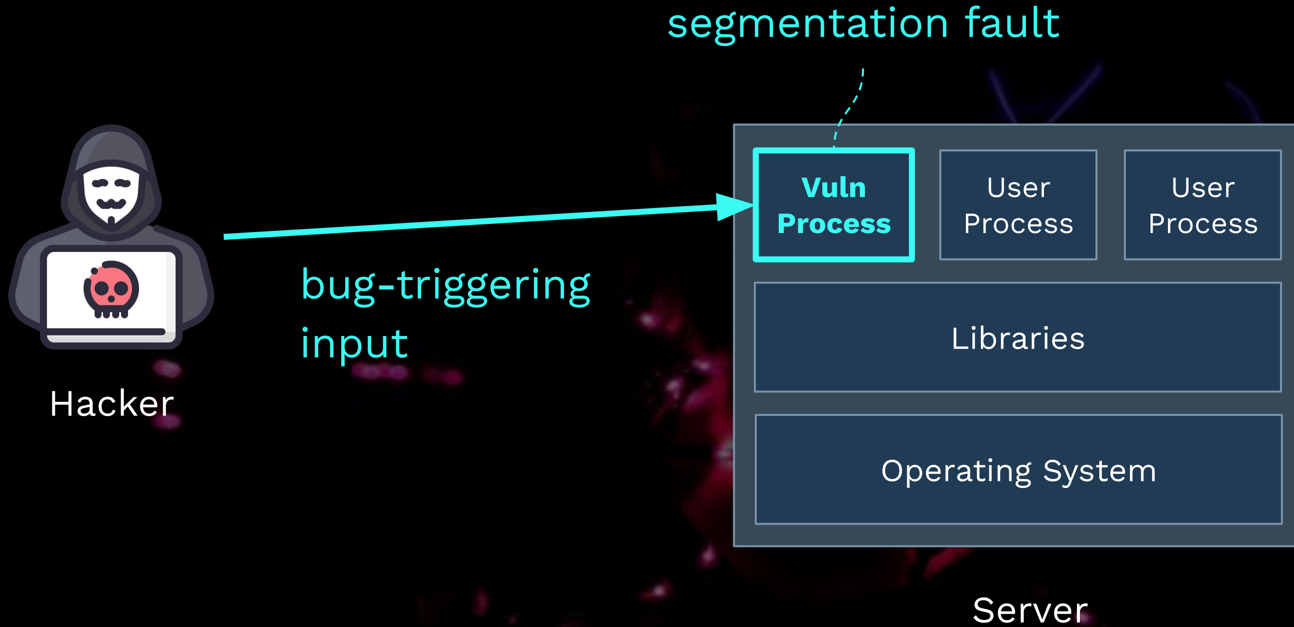
01

Introduction

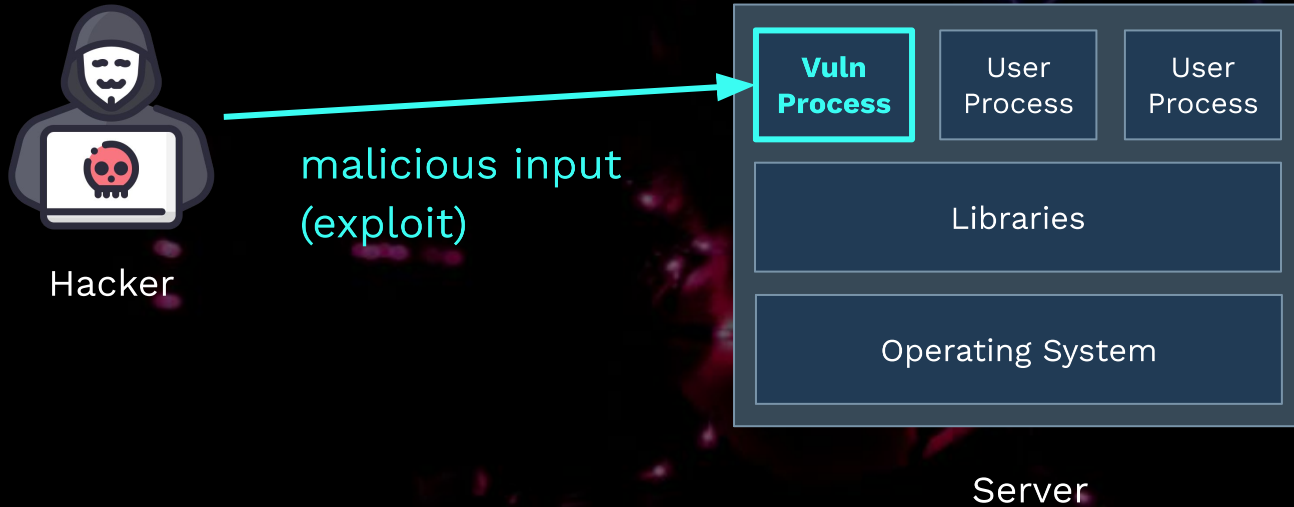
0x11 Introduction



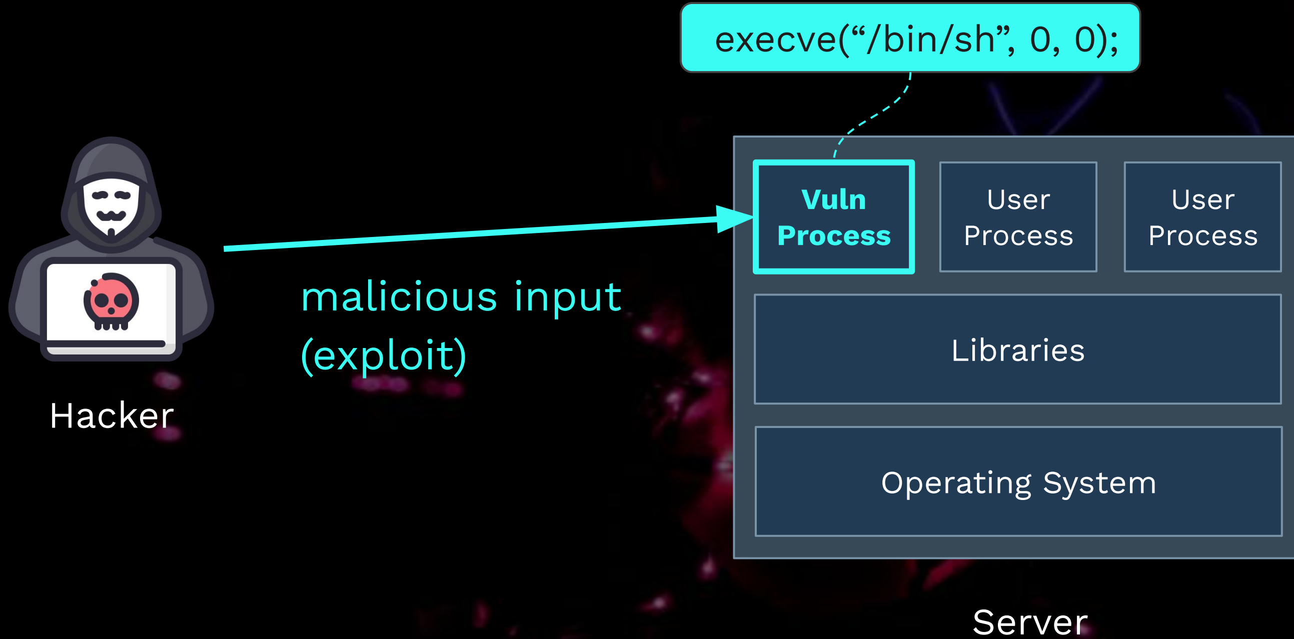
0x11 Introduction



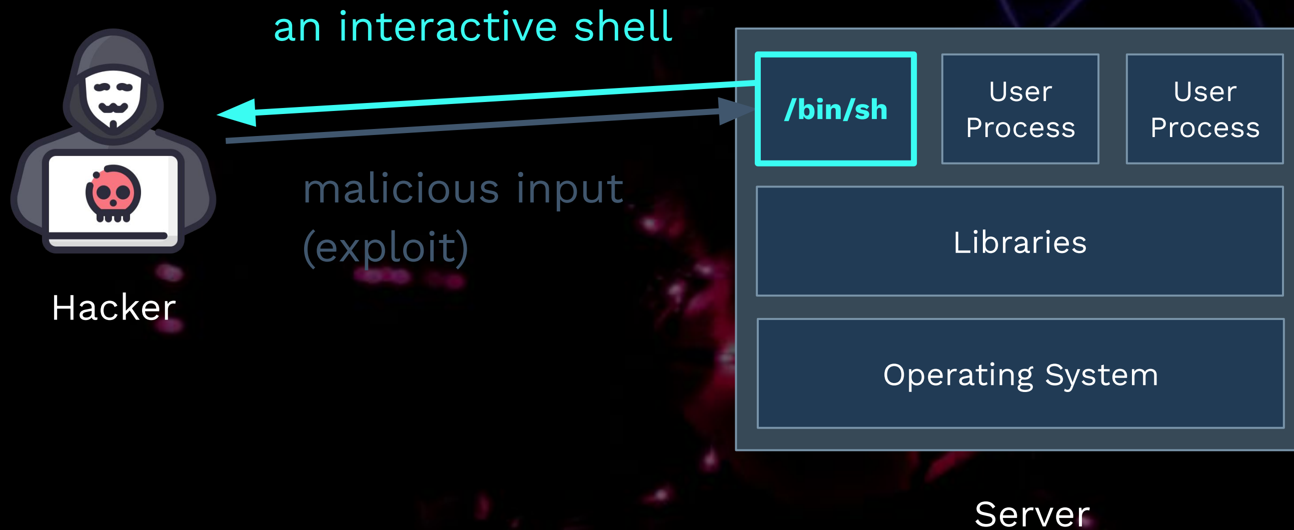
0x11 Introduction



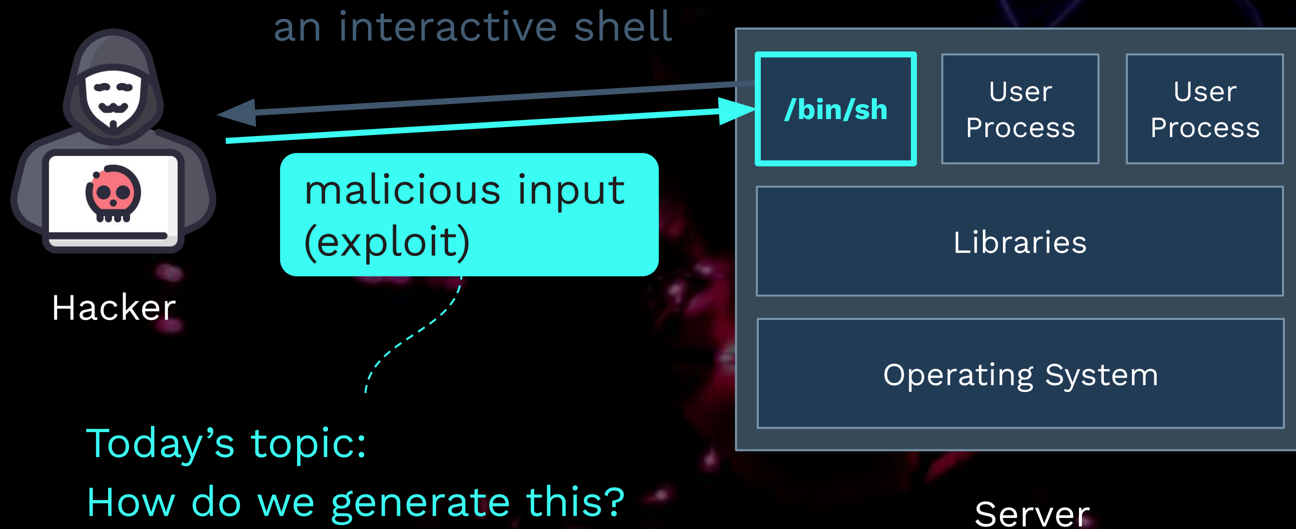
0x11 Introduction



0x11 Introduction



0x11 Introduction



0x12 **Definitions**

- Exploit
  - [vt.] To take advantage of a vulnerability in a program.
  - [n.] A chunk of **data** (i.e. **payload**) that “exploits” the vulnerability.
- Exploit Script
  - E.g., a python script which uses pwntools to interact with the vuln. process.
- Results
  - Arbitrary code execution, auth bypassing, privesc, etc.

0x13 Past Research

Table Past Research on Automatic Exploit Generation (Selected)

	AEG (2011)	MAYHEM (2012)	CRAX (2014)	Revery (2018)	LAEG (2021)
Developer(s)	CMU	CMU	SQLab, NCTU	CAS, UCAS, Tsinghua University (Beijing)	NSLab, NTU
Paper	CACM (2014)	USENIX Security Symposium (2011)	IEEE Transactions on Reliability (2014)	ACM SIGSAC (2018)	PASS4IOT (2022)
Vuln. Types	Stack Overflow, Fmt	Stack Overflow, Fmt	Stack/Heap Overflow Fmt, Uninitialized Vars	Heap Overflow Double Free, UAF	Stack Overflow, Information Leak
Based on	-	-	S^2E 1.X	AFL, angr	Qiling
Method	1. Find bugs from LLVM IR 2. Exploit constraint: symbex	1. Hybrid symbex 2. Selective path	Selective code/path/input	1. Fuzz diverging paths 2. Symbex for path stitching	1. Dynamic taint analysis 2. I/O states analysis
Bypass Prot.	-	-	-	NX	ASLR, NX, PIE, Canary
Scale	Xmail	dizzy	Microsoft Word, MPlayer, Foxit PDF Reader	CTF	CTF
Open Source	No	No (Commercial)	Yes	No	No

0x14 CRAX (2014)

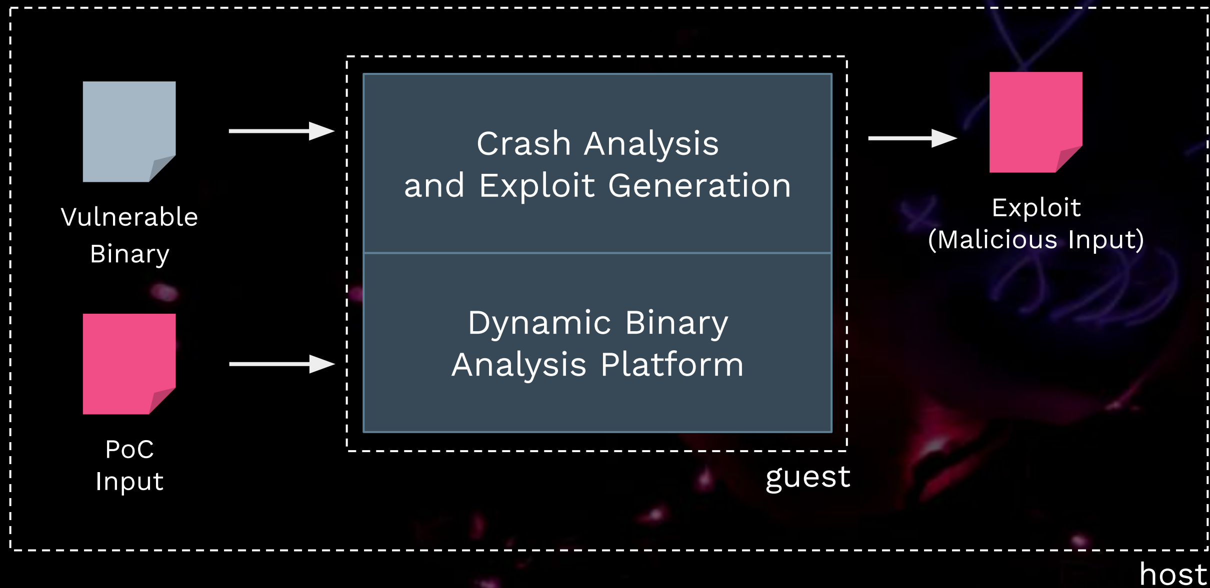
- CRAX = Software CRash analysis for Automatic eXploit Generation
  - Successfully exploited
    - **Microsoft Office** (CVE-2010-3333, CVE-2012-0158)
    - **Mplayer** (CVE-2008-0630, EDB-ID-17013)
  - Bypass protections?
    - all protections disabled
  - Platform / Method
    - **S^2E** 1.X / selective symbolic execution

0x15 **LAEG (2021)**

- LAEG = Leak-based AEG
  - Successfully exploited
    - DEFCON’27 CTF speedrun-00{1,2}
    - ångstromCTF 2020 no_canary, 2021 tranquil
  - Bypass protections?
    - using information leak, it can **bypass ASLR, NX, PIE and Canary**
  - Platform / Method
    - **Qiling Framework** / dynamic taint analysis + **I/O States analysis**

Background

0x21 Preparing Tools



0x21 Preparing Tools

- A **dynamic binary analysis platform** which provides ...
    - API to r/w guest register and memory
    - Virtual memory map
    - Runtime instrumentation (e.g., ❶ Intel Pin, ❷ Instruction and syscall hooks)
    - Symbolic execution
    - Handles system calls reliably
- ELF parsing library (optional)
    - e.g., LIEF, pwntools

0x21 Preparing Tools

- A dynar
  - API t
  - Virtu
  - Runt ... l hooks)
  - Hanc
  - Sym
- ELF par
  - e.g., LIEF, pwntools

```
18 [State 0] CRAX: Dumping memory map...
---------------- [VMMAP] ----------------
Start           End             Perm    Module
0x55c0fb747000  0x55c0fb748000  r--     target
0x55c0fb748000  0x55c0fb749000  r-x     target
0x55c0fb749000  0x55c0fb74b000  r--     target
0x55c0fb74b000  0x55c0fb74c000  rw-     target
0x7f6dddc2f000  0x7f6ddddc4000  r-x     libc.so.6
0x7f6ddddc4000  0x7f6dddfc4000  ---     libc.so.6
0x7f6dddfc4000  0x7f6dddfc8000  r--     libc.so.6
0x7f6dddfc8000  0x7f6dddfce000  rw-     libc.so.6
0x7f6dddfce000  0x7f6dddff1000  r-x     ld-linux-x86-64.so.2
0x7f6dde1e7000  0x7f6dde1e9000  rw-     ld-linux-x86-64.so.2
0x7f6dde1f1000  0x7f6dde1f2000  r--     ld-linux-x86-64.so.2
0x7f6dde1f2000  0x7f6dde1f3000  rw-     ld-linux-x86-64.so.2
0x7fff25e84000  0x7fff25e86000  rw-     [stack]
```

0x21 Preparing Tools

- A **dynamic binary analysis platform** which provides ...
  - API to r/w guest register and memory
  - Virtual memory map
  - Runtime instrumentation (e.g., ❶ Intel Pin, ❷ Instruction and syscall hooks)
  - Symbolic execution
  - **Handles system calls reliably**
- ELF parsing library (optional)
  - e.g., LIEF, pwntools

0x21 Preparing Tools

Table Comparison of Dynamic Binary Analysis Platform

	KLEE (2008)	S^2E (2011)	Triton (2015)	angr (2016)	Qiling Framework (2019)
Supported Arch.	x86, x86_64	x86, x86_64	x86, x86_64, ARM, ARM64	Any arch. supported by Valgrind	Any arch. supported by Unicorn
Languages	C/C++14	C/C++17, Lua	C/C++14	Python 3	Python 3
Program Execution	Interprets LLVM bitcode	Virtualization (qemu-kvm) + KLEE	Intel Pin	SimEngines	Unicorn
System Calls Emulation	Partial (KLEE-uClibc)	Full (Virtualization)	No	Partial (Emulated)	Partial (Emulated)
Runtime Instrumentation	No	Not Supported Directly	Yes	Yes	Yes
Symbolic Execution	Yes	Yes	Yes	Yes	No
Dynamic Taint Analysis	Yes (symbolic taint)	Yes (symbolic taint)	Yes	Yes (symbolic taint)	No

0x22 Dynamic Binary Analysis

- Symbolic Execution

Example Program

```
1  void func(int y) {
2      int z = y * 2;
3
4      if (z > 12) {
5          if (y < 10) {
6              system("/bin/sh");
7          } else {
8              printf("?_?");
9          }
10     } else {
11         printf("Failed");
12     }
13 }
```

0x22 Dynamic Binary Analysis

- Symbolic Execution

From Wikipedia:

Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute.

```
        } else {
            printf("?_?");
        }
    } else {
        printf("Failed");
    }
}
```

0x22 Dynamic Binary Analysis

- Symbolic Execution

Example Program

```
1  void func(int y) {
2      int z = y * 2;
3
4      if (z > 12) {
5          if (y < 10) {
6              system("/bin/sh");
7          } else {
8              printf("?_?");
9          }
10     } else {
11         printf("Failed");
12     }
13 }
```

Input: y

Q: How will y affect program execution?

→ Make y symbolic

0x22 Dynamic Binary Analysis

- Symbolic Execution

Example Program

```
1  void func(int y) {
2      int z = y * 2;
3
4      if (z > 12) {
5          if (y < 10) {
6              system("/bin/sh");
7          } else {
8              printf("?_?");
9          }
10     } else {
11         printf("Failed");
12     }
13 }
```

Input: y

Q: How will y affect program execution?

→ Make y symbolic

→ z = 2y (z is now also symbolic)

0x22 Dynamic Binary Analysis

- Symbolic Execution

Example Program

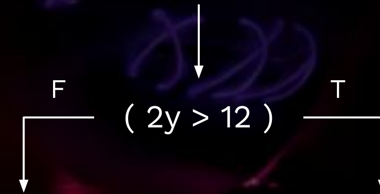
```
1  void func(int y) {
2      int z = y * 2;
3
4      if (z > 12) {
5          if (y < 10) {
6              system("/bin/sh");
7          } else {
8              printf("?_?");
9          }
10     } else {
11         printf("Failed");
12     }
13 }
```

Input: y

Q: How will y affect program execution?

→ Make y symbolic

→ z = 2y (z is now also symbolic)



0x22 Dynamic Binary Analysis

- Symbolic Execution

Example Program

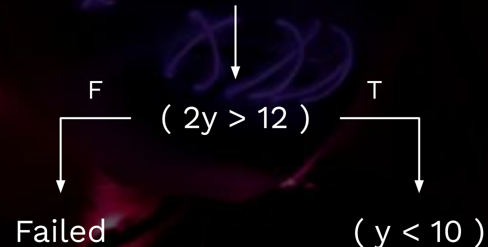
```
1  void func(int y) {
2      int z = y * 2;
3
4      if (z > 12) {
5          if (y < 10) {
6              system("/bin/sh");
7          } else {
8              printf("?_?");
9          }
10     } else {
11         printf("Failed");
12     }
13 }
```

Input: y

Q: How will y affect program execution?

→ Make y symbolic

→ z = 2y (z is now also symbolic)



0x22 Dynamic Binary Analysis

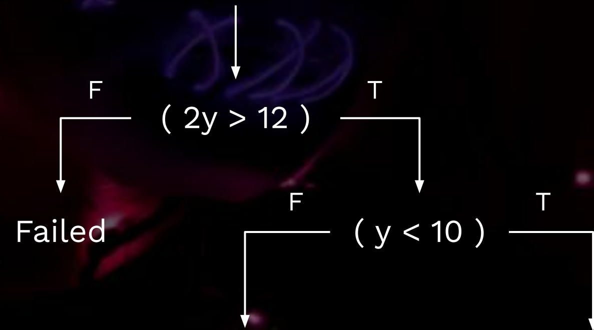
- Symbolic Execution

Example Program

```
1  void func(int y) {
2      int z = y * 2;
3
4      if (z > 12) {
5          if (y < 10) {
6              system("/bin/sh");
7          } else {
8              printf("?_?");
9          }
10     } else {
11         printf("Failed");
12     }
13 }
```

Input: y

Q: How will y affect program execution?
→ Make y symbolic
→ z = 2y (z is now also symbolic)



0x22 Dynamic Binary Analysis

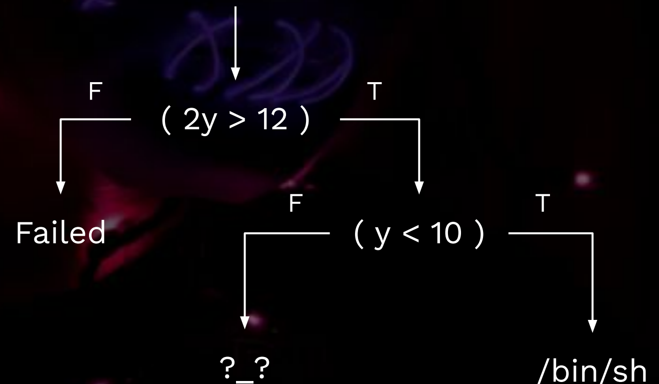
- Symbolic Execution

Example Program

```
1  void func(int y) {
2      int z = y * 2;
3
4      if (z > 12) {
5          if (y < 10) {
6              system("/bin/sh");
7          } else {
8              printf("?_?");
9          }
10     } else {
11         printf("Failed");
12     }
13 }
```

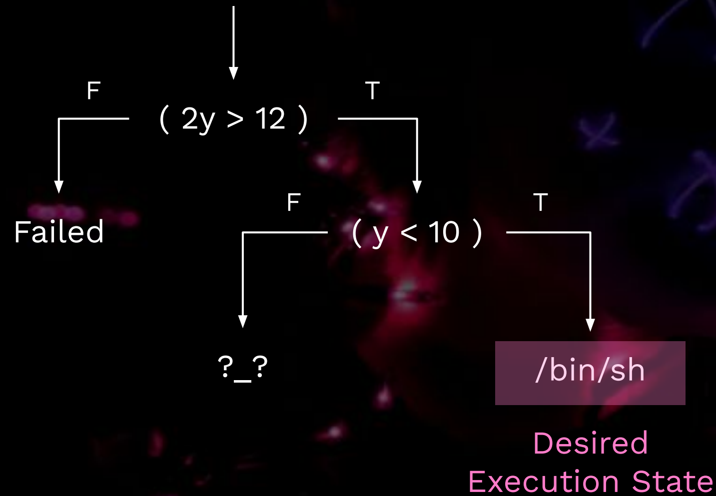
Input: y

Q: How will y affect program execution?
→ Make y symbolic
→ z = 2y (z is now also symbolic)



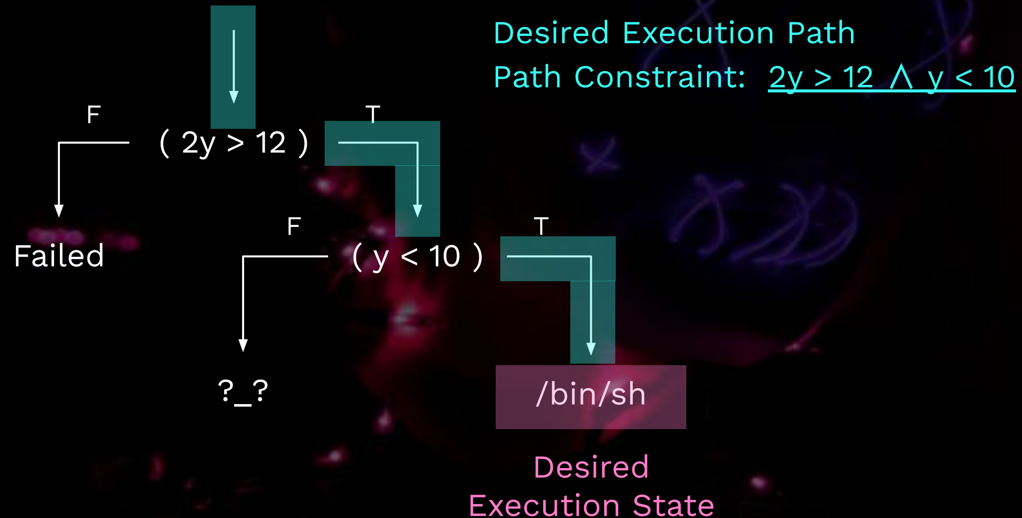
0x22 Dynamic Binary Analysis

- Symbolic Execution



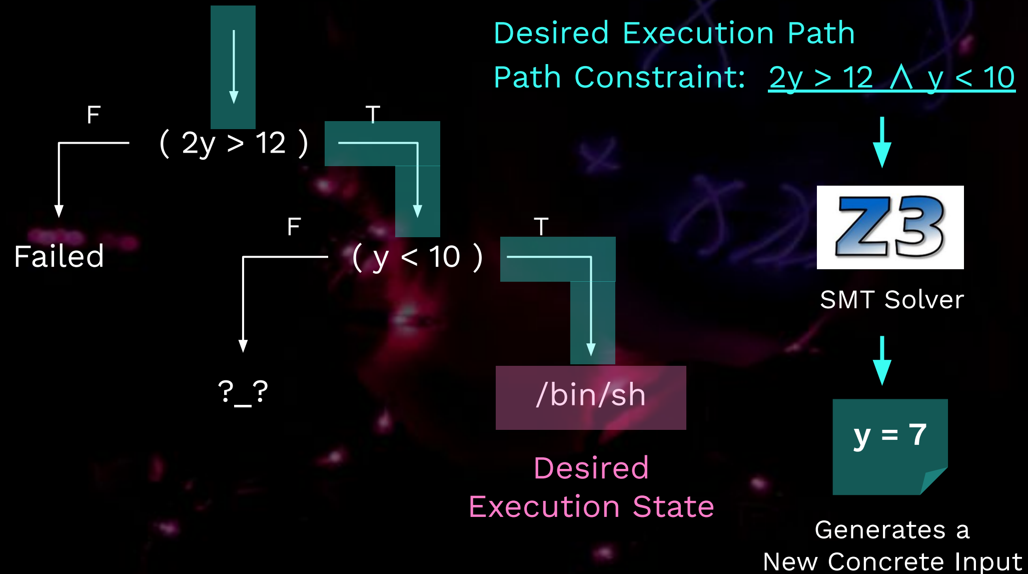
0x22 Dynamic Binary Analysis

- Symbolic Execution



0x22 Dynamic Binary Analysis

- Symbolic Execution



0x22 Dynamic Binary Analysis

- Symbolic Execution
  - Whenever we execute a branch instruction, the engine forks state.
  - Explores all execution paths in a single run.
  - If the target binary is large → Lots of paths to explore → "**Path Explosion**" (2^n)
- Dynamic Symbolic Execution
  - = Selective Symbolic Execution = Concolic Execution
  - Don't fork states upon branches. Collects path constraints only.
  - Explores only one path in a single run, and generate a new input.

0x22 Dynamic Binary Analysis

- Symbolic Execution
  - Whenever we execute a branch instruction, the engine forks state.
  - Explores all execution paths in a single run.
  - If the target binary is large → Lots of paths to explore → **"Path Explosion"** (2^n)
- Dynamic Symbolic Execution
  - = Selective Symbolic Execution = Concolic Execution
  - Don't fork states upon branches. Collects path constraints only.
  - Explores only one path in a single run, and generate a new input.

0x22 Dynamic Binary Analysis

- Properties of symbolic execution
    - Symbolic bytes are infectious

      ❶ Let RDX be symbolic

      ❷ `mov QWORD PTR[0x403010], RDX` // QWORD at 0x403010 is now symbolic.

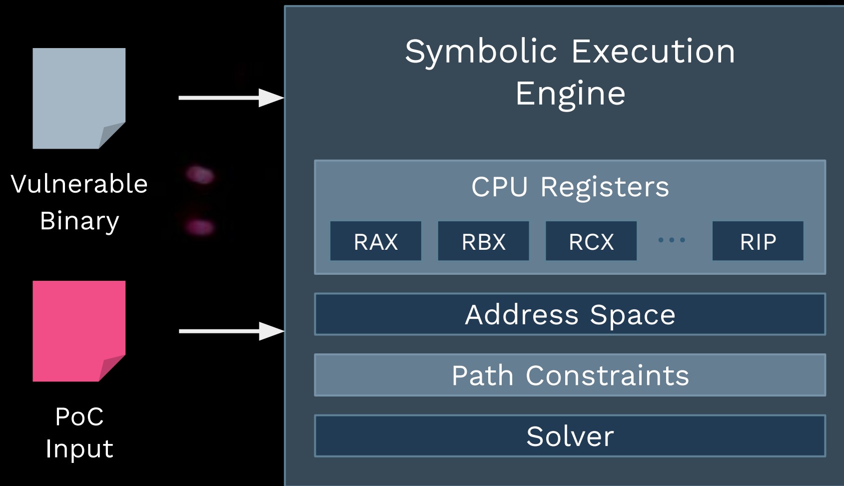
      ❸ `mov RCX, QWORD PTR[0x403010]` // RCX is now symbolic.

    - Usage of solver
        - Test case (input) generation
        - Exploit generation

CRAXplusplus

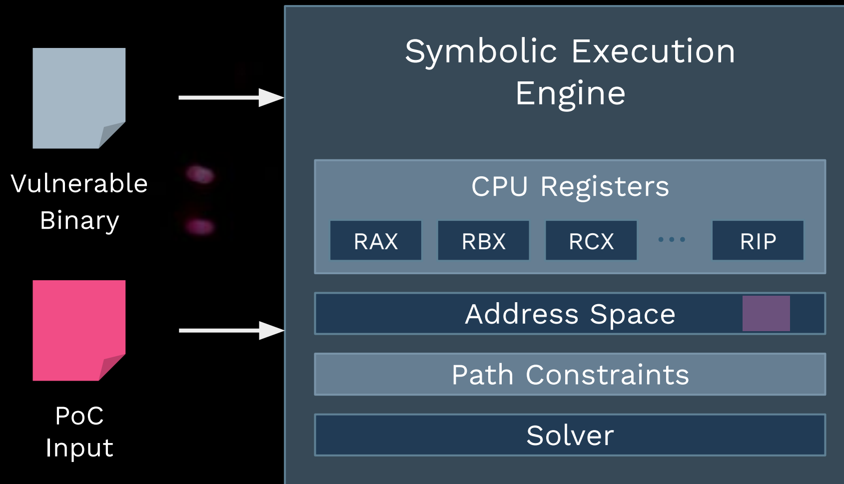
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



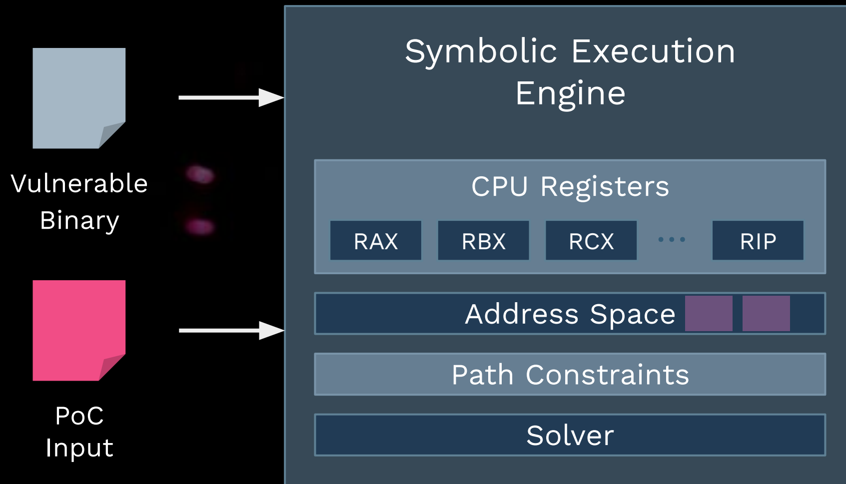
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



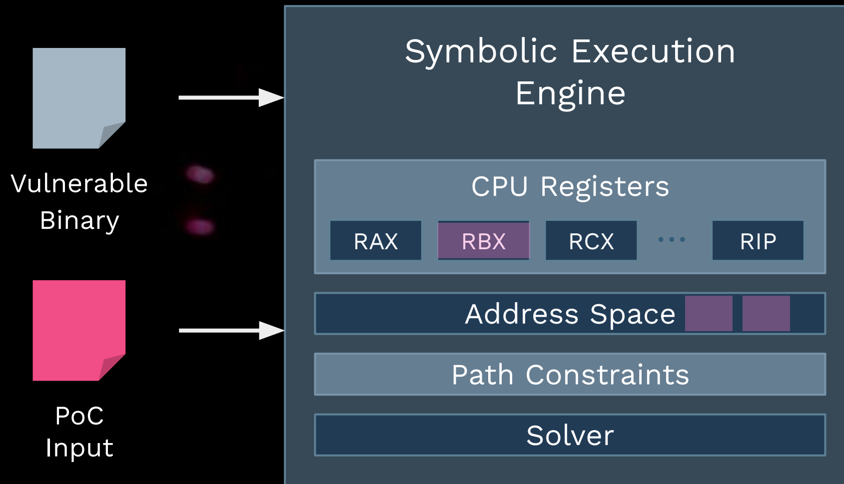
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



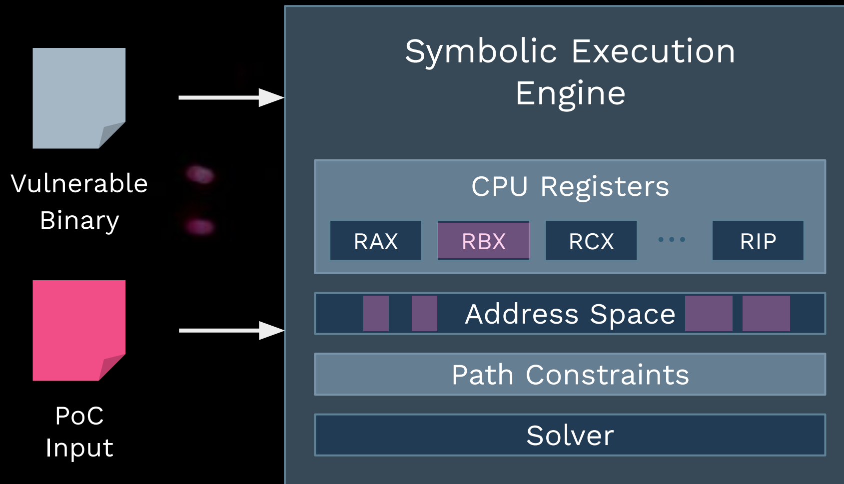
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



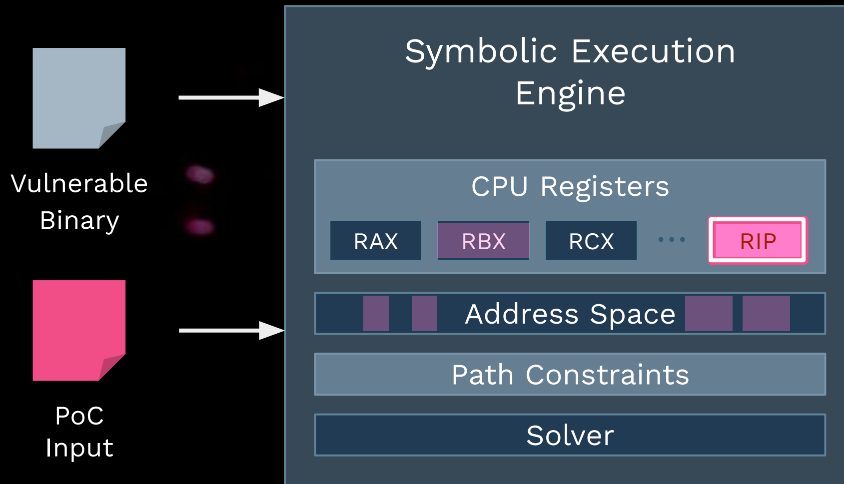
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



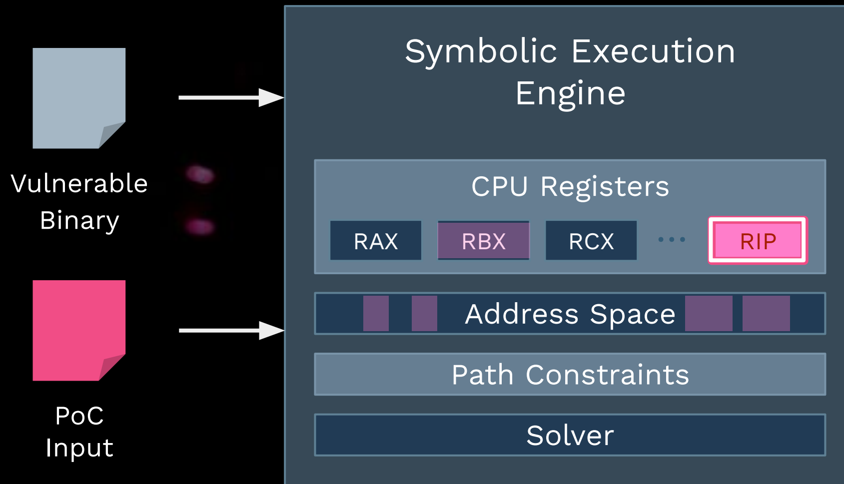
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



0x31 Exploit Generation

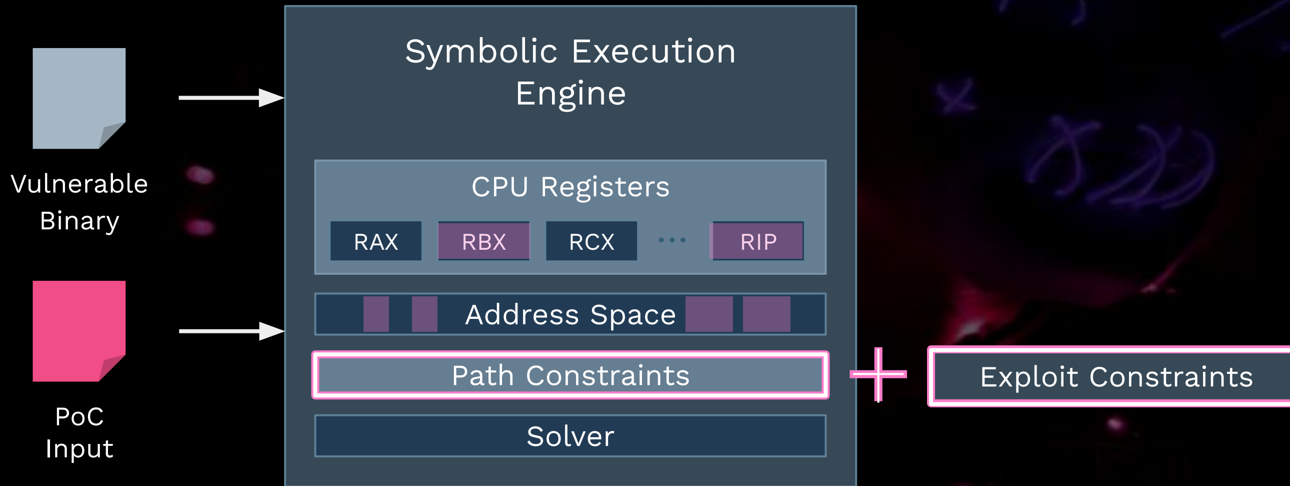
- Symbolic Execution and Exploit Generation



Symbolic RIP
(segmentation fault)

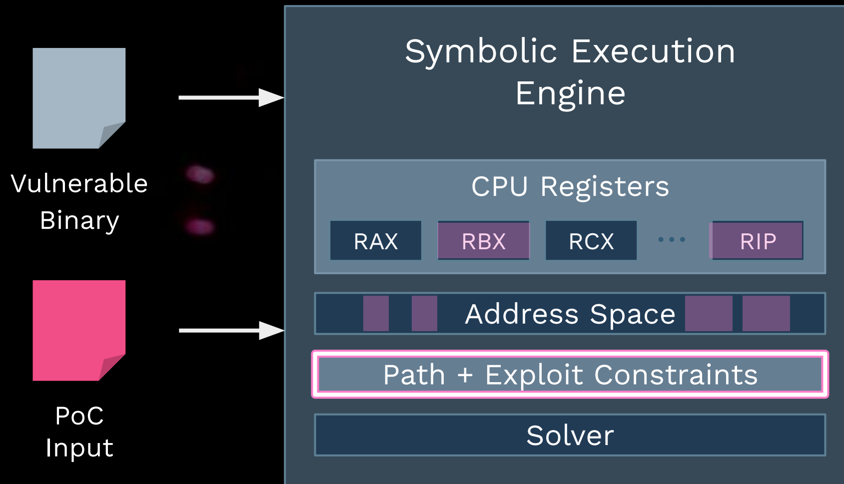
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



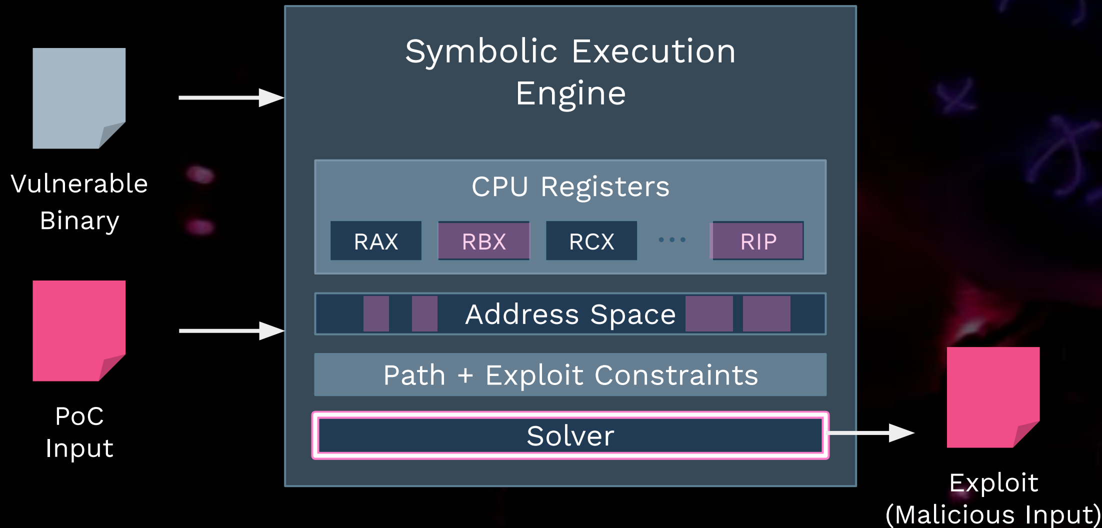
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



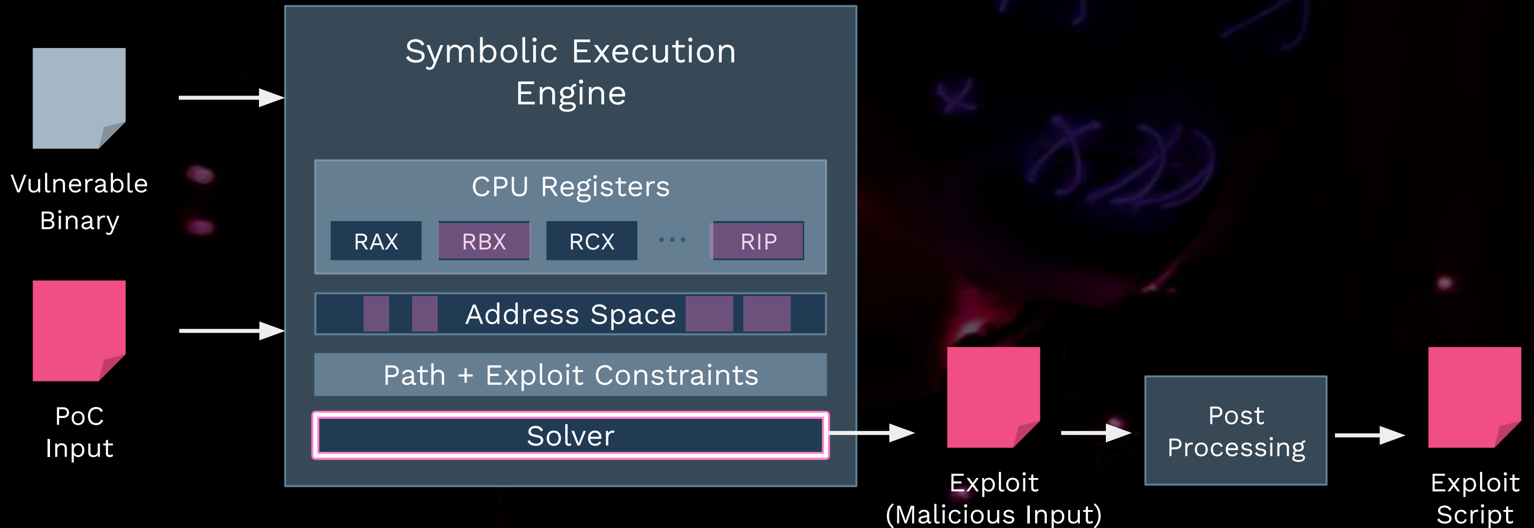
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



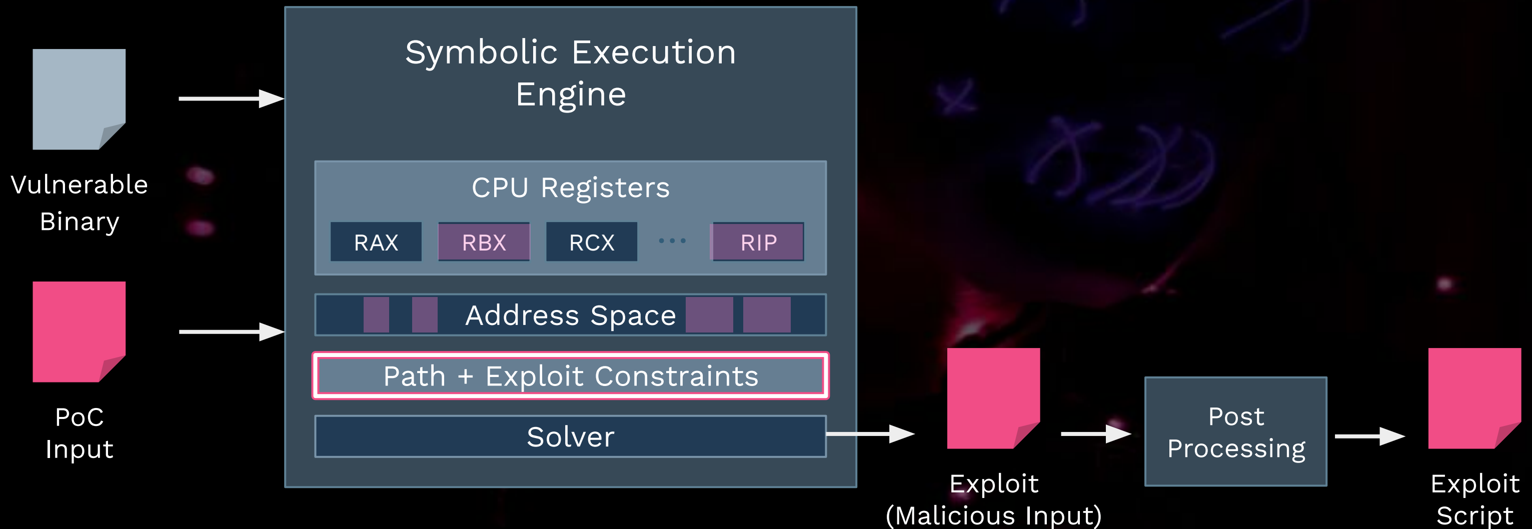
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



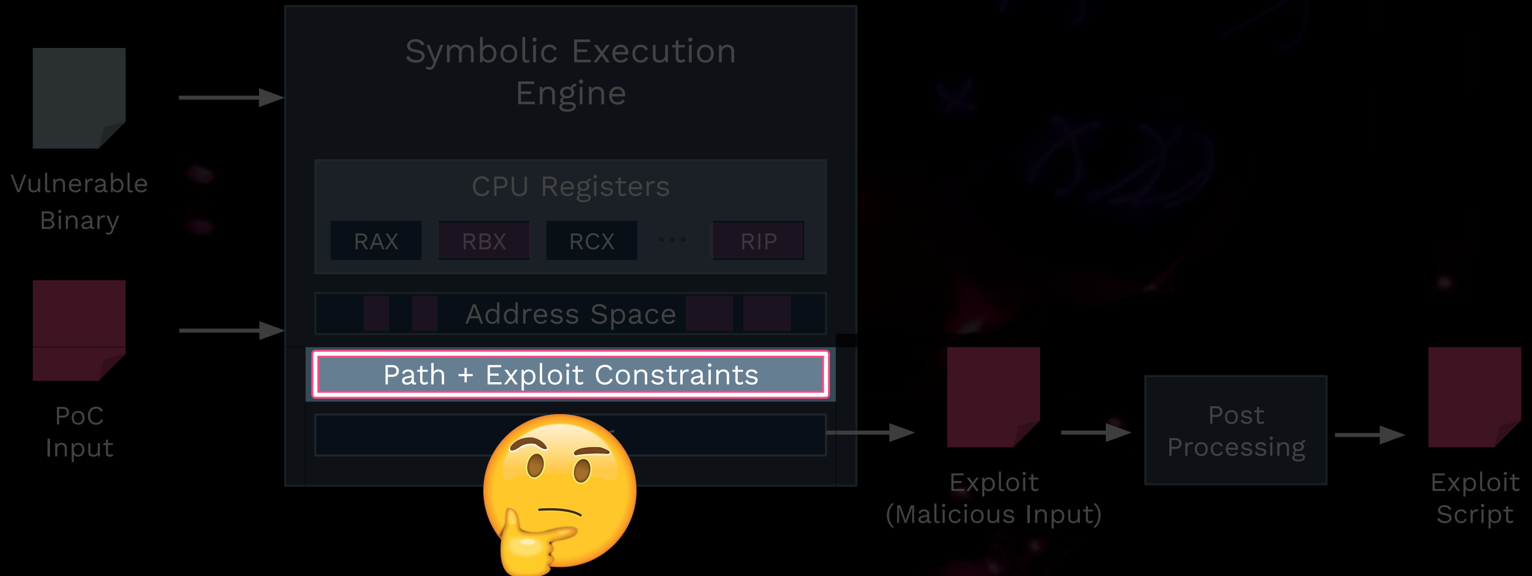
0x31 Exploit Generation

- Symbolic Execution and Exploit Generation



0x31 Exploit Generation

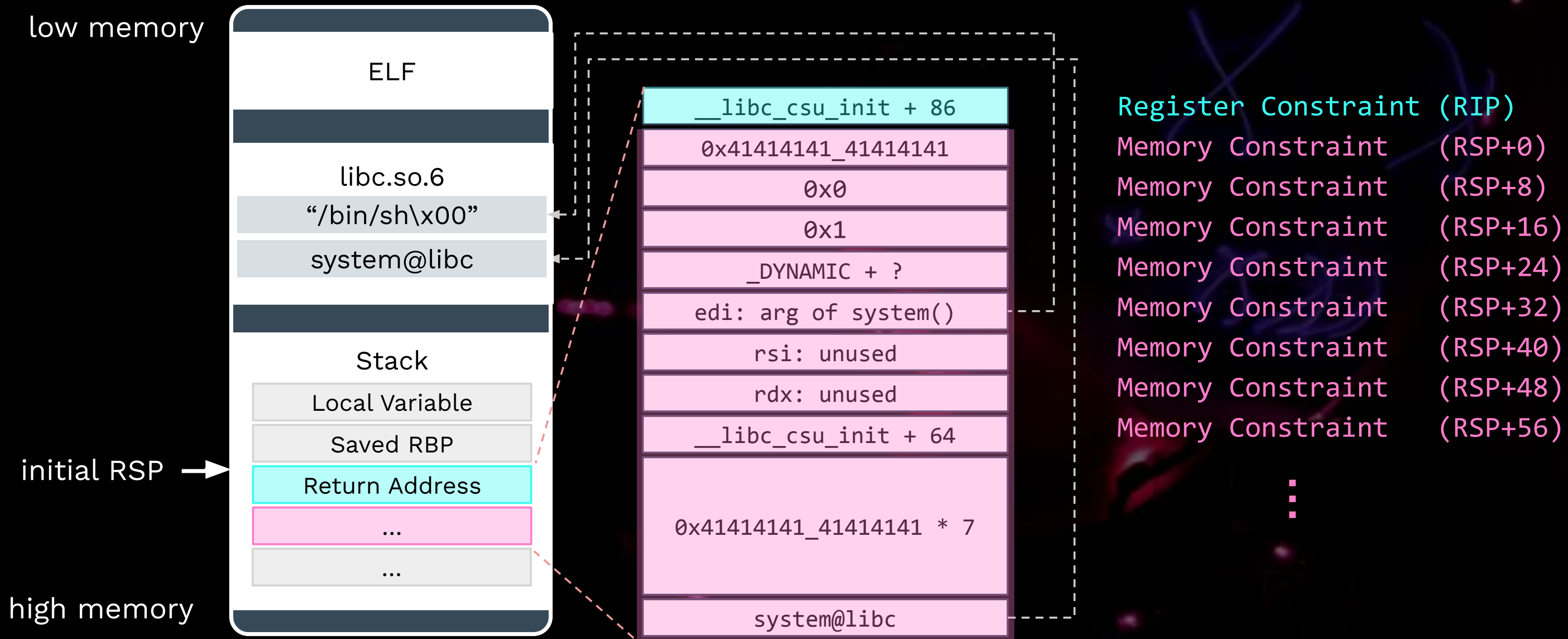
- Symbolic Execution and Exploit Generation



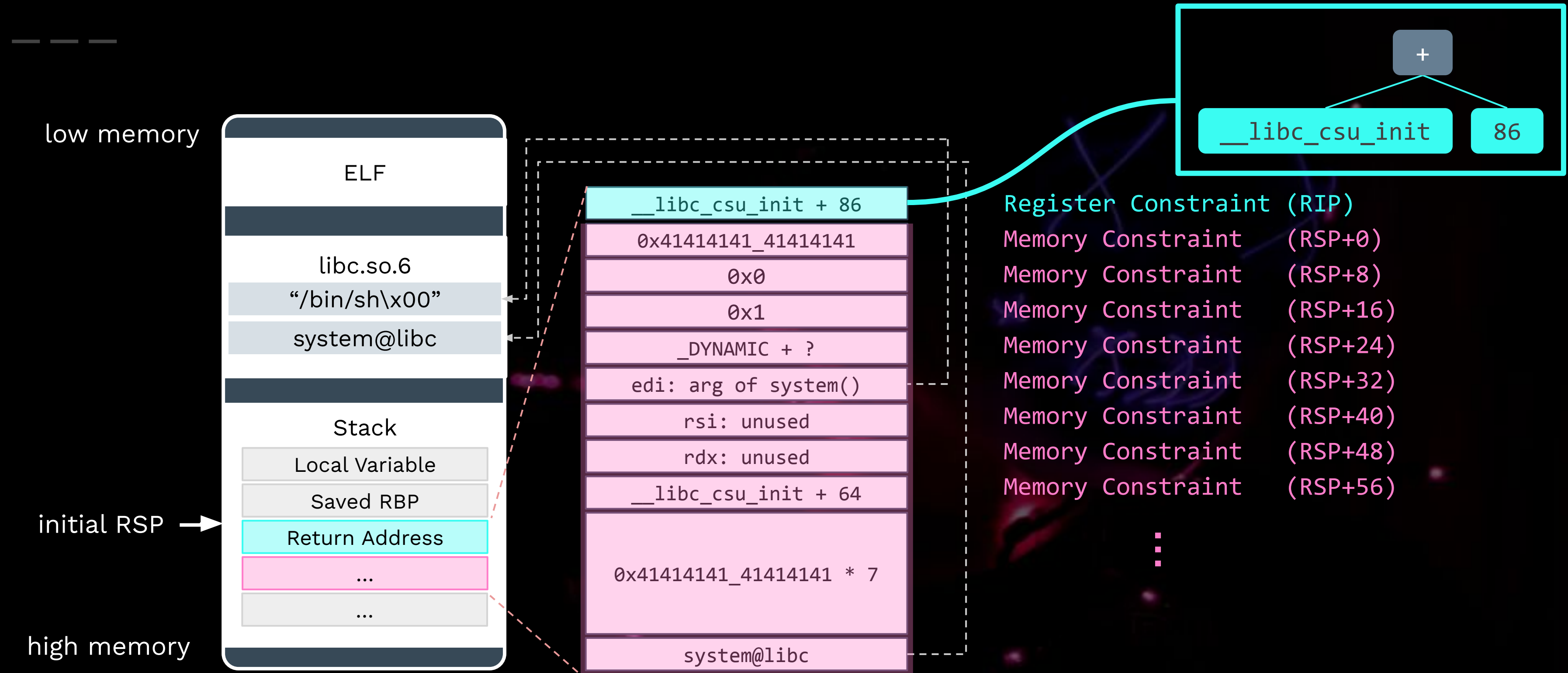
0x31 Exploit Generation

- We define two types of exploit constraints
  - **Register** constraints
  - **Memory** constraints
- Examples
  - e.g., Register: `RIP = 0xcafe'babe'dead'beef`
- Solver
  - gives an input which crashes with `RIP = 0xcafe'babe'dead'beef`

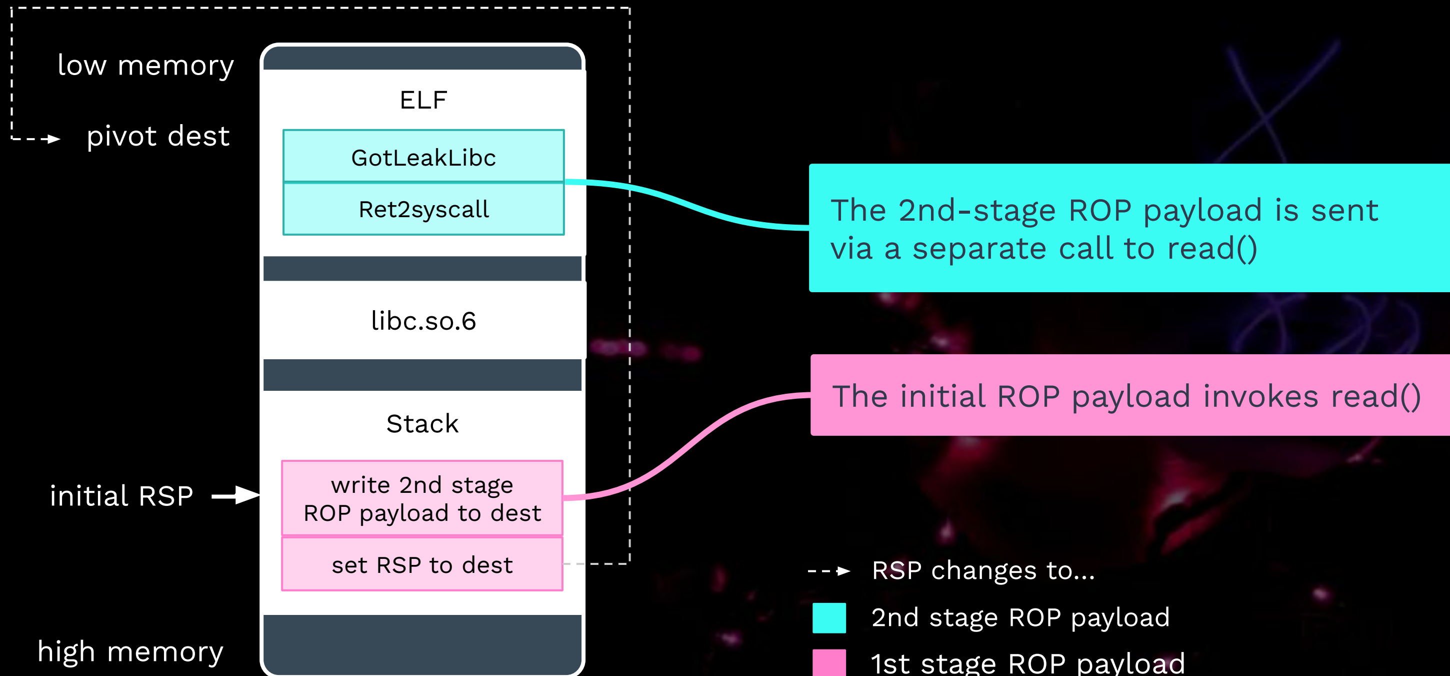
0x32 Example: system("/bin/sh") via Ret2csu



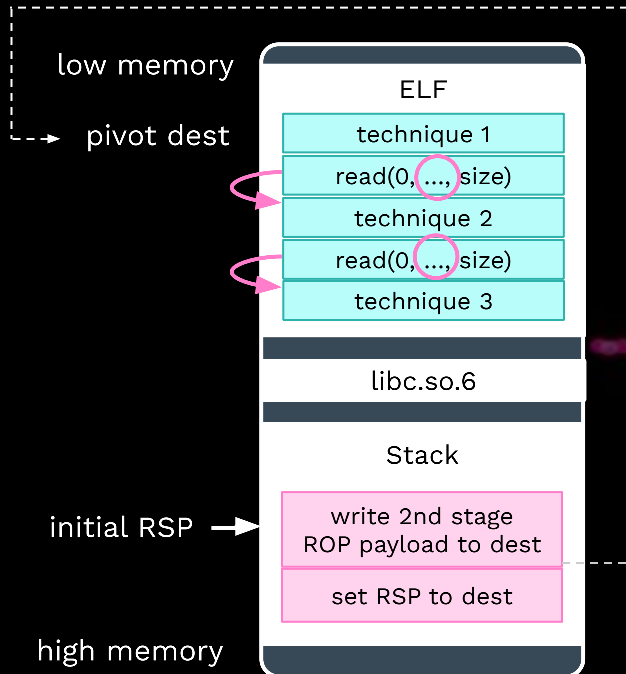
0x32 Example: system("/bin/sh") via Ret2csu



0x32 Example: Stack Pivoting



0x32 Example: Multi-Technique Chaining



Invokes read() multiple times and "glue" techniques together

read() destinations must be generated precisely.

Currently read() and gets() are supported in this regard.

- read() and write to...
- RSP changes to...
- 2nd stage ROP payload
- 1st stage ROP payload

0x33 ROP Payload Builder

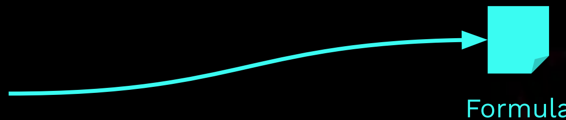
- ROP Payload Builder
  - **Purpose**
    - each technique has a ROP payload formula
    - ROP payload builder merges them into a single one
  - **Symbolic Mode**
    - addRegisterConstraint() - constrains a register x to have value y.
    - addMemoryConstraint() - constrains a memory location m to have value n.
    - getOneConcreteInput() - query the solver for a concrete input (std::vector<uint8_t>)
  - **Direct Mode**
    - no solver involved
    - statically concats ROP payloads

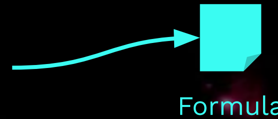
0x34 Exploitation Techniques

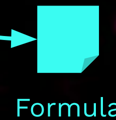
- Techniques
  - Ret2csu
  - Basic Stack Pivoting
  - Advanced Stack Pivoting
  - GOT Leak Libc
  - Ret2syscall
  - One Gadget

0x34 Exploitation Techniques

- Techniques
  - Ret2csu
  - Basic Stack Pivoting
  - Advanced Stack Pivoting
  - GOT Leak Libc
  - Ret2syscall
  - One Gadget

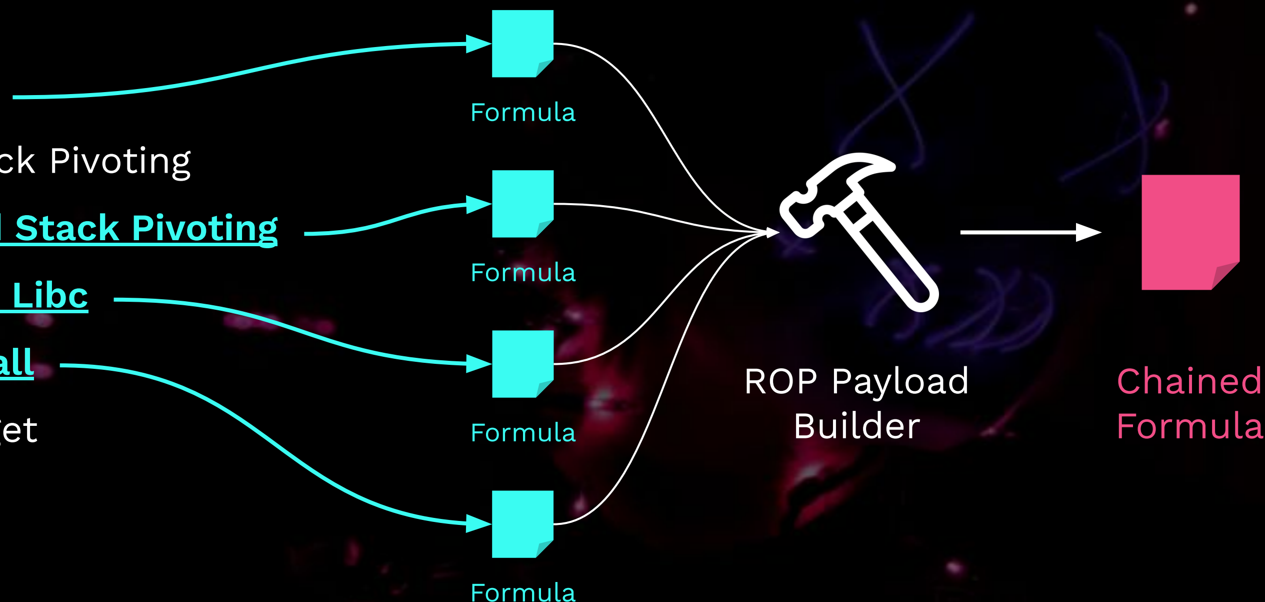






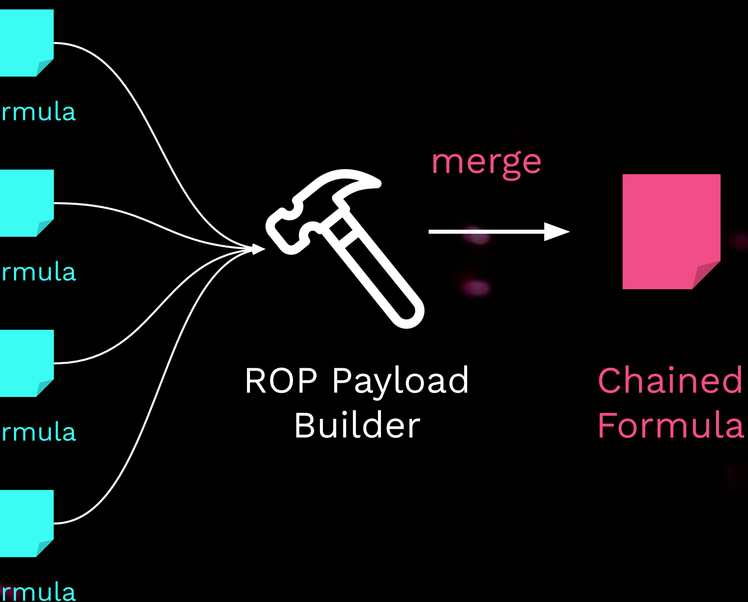
0x34 Exploitation Techniques

- Techniques
  - Ret2csu
  - Basic Stack Pivoting
  - Advanced Stack Pivoting
  - GOT Leak Libc
  - Ret2syscall
  - One Gadget



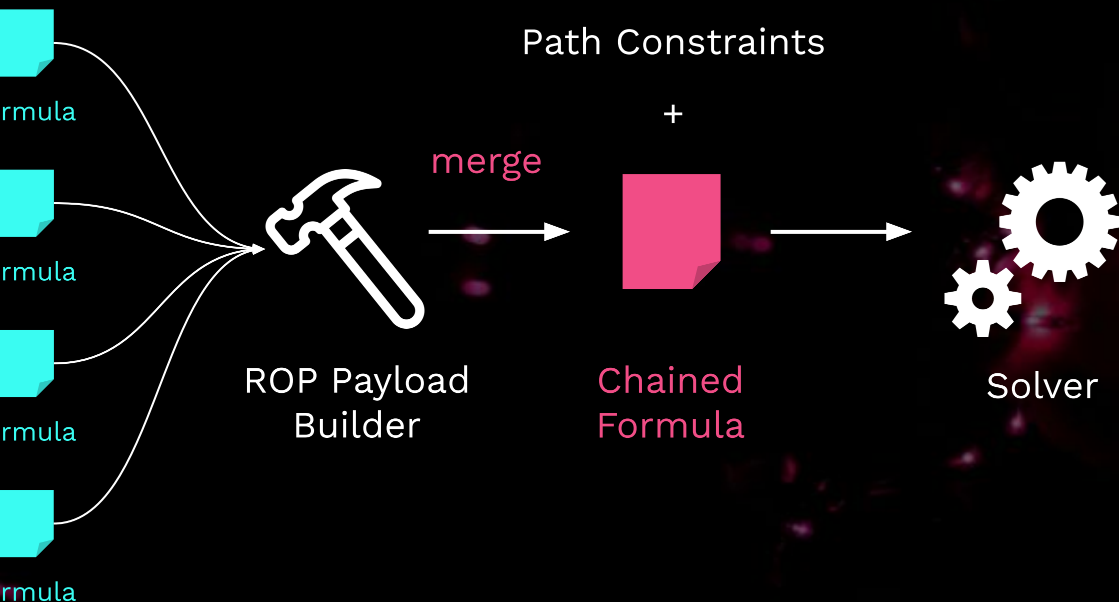
0x34 Exploitation Techniques

- Solver



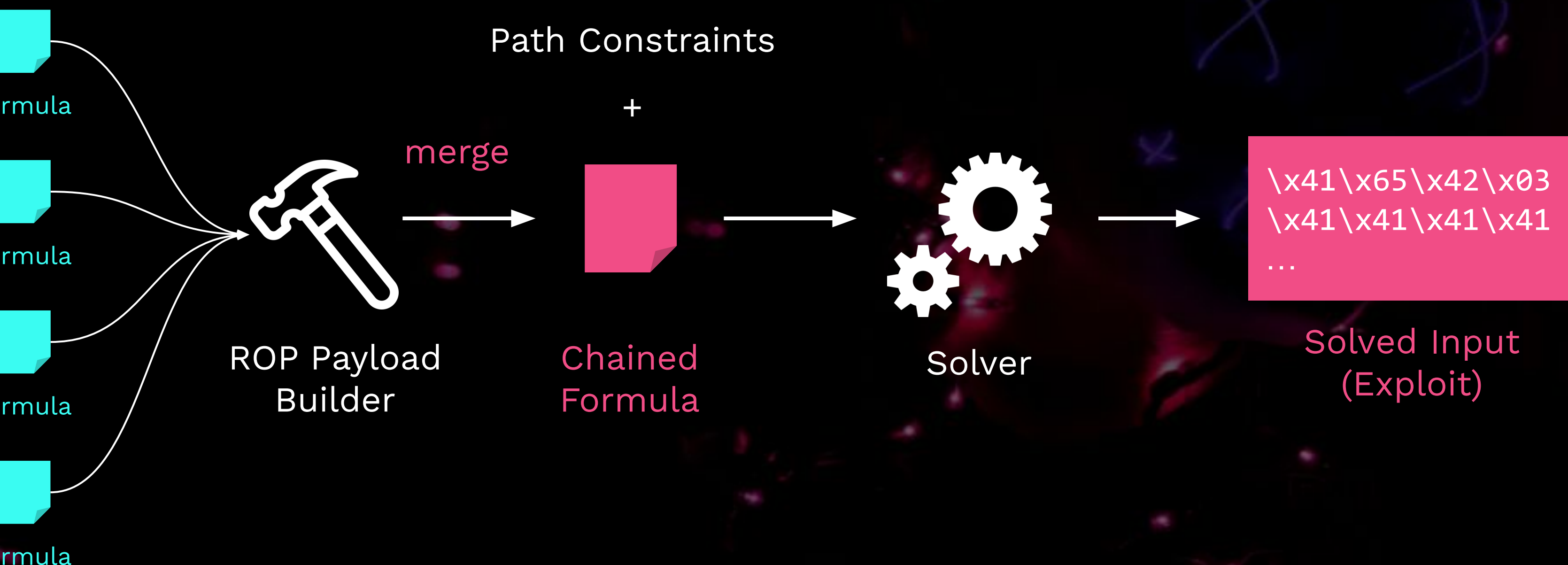
0x34 Exploitation Techniques

- Solver



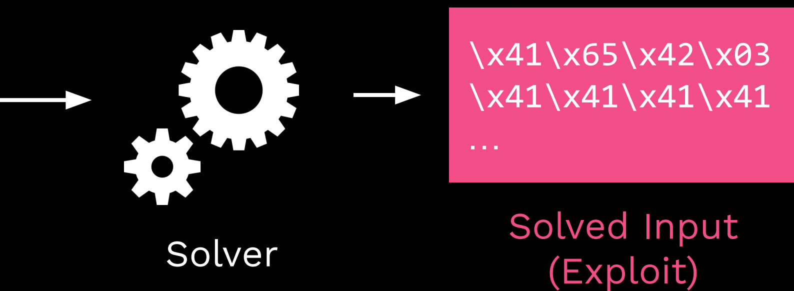
0x34 Exploitation Techniques

- Solver



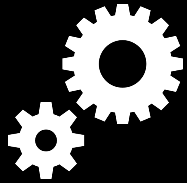
0x34 Exploitation Techniques

- Post Processing



0x34 Exp. Techniques

- Post Processing



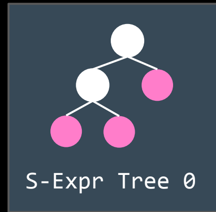
\x41\x65\x42\x03
\x41\x41\x41\x41
...

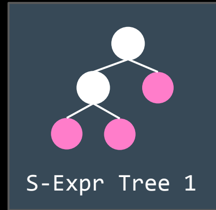
Solved Input
(Exploit)

```python
#!/usr/bin/env python3
from pwn import *
context.update(arch = 'amd64', os = 'linux', log_level = 'info')

target = ELF('./target', checksec=False)
libc_2_24_so = ELF('./libc-2.24.so', checksec=False)

__libc_csu_init = 0x400840
__libc_csu_init_call_target = 0x400e48
__libc_csu_init_gadget1 = 0x400896
__libc_csu_init_gadget2 = 0x400880
canary = 0x0
libc_2_24_so_base = 0x0
pivot_dest = 0x601860
target_base = 0x0
target_leave_ret = 0x40074a
target_pop_rbp_ret = 0x400668

if __name__ == '__main__':
    proc = process(['./ld-2.24.so', './target'], env={'LD_PRELOAD': './libc-2.24.so'})
    payload  = b'\x45\x76\x65\x72\x79\x74\x68\x69\x6e\x67\x20\x69\x6e\x74\x65\x6c\x6c\x6
    proc.send(payload)
    time.sleep(0.2)

    proc.recvrepeat(0)
    payload  = p64(0x0)
    payload += p64(target_base + __libc_csu_init_gadget1)
    payload += p64(0x4141414141414141)
    payload += p64(0x0)
    payload += p64(0x1)
    payload += p64(target_base + __libc_csu_init_call_target)
    payload += p64(0x0)
    payload += p64(target_base + target.got['read'])
    payload += p64(0x1)
    payload += p64(target_base + __libc_csu_init_gadget2)
```

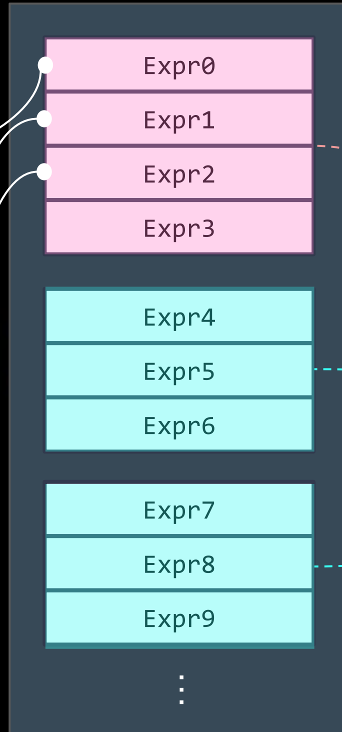
0x34 ROP Payload Builder

S-Expr Trees, each of which represents an expression.





ROP payload formula
(a 2D list of trees)



Generated exploit script

```
target = ELF('target')

if __name__ == '__main__':
    proc = target.process()

    payload  = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00...'
    proc.send(payload)
    time.sleep(0.2)
```
generated in symbolic mode

```
    payload  = p64(Expr4)
    payload += p64(Expr5)
    payload += p64(Expr6)
    proc.send(payload)
    time.sleep(0.2)
```
generated in direct mode

```
    payload  = p64(Expr7)
    payload += p64(Expr8)
    payload += p64(Expr9)
    proc.send(payload)
    time.sleep(0.2)
```
generated in direct mode

```
    ...
    proc.interactive()
```

0x35 Extending CRAX++

CRAX++ Config

```
...
pluginsConfig.CRAX = {
    showInstructions = false,
    showSyscalls = true,
    concolicMode = true,

    modules = {
        "CodeSelection",   -- CRAX   (2014)
        "IOStates",        -- LAEG   (2021)
        "DynamicRop"       -- CRAX++ (2022)
    },

    techniques = {
        "Ret2csu",
        "AdvancedStackPivoting",
        "GotLeakLibc",
        "Ret2syscall"
    },
}
```

0x35 Extending CRAX++

- Techniques
  - Ret2stack
  - Ret2csu
  - Basic Stack Pivoting
  - Advanced Stack Pivoting
  - GOT Leak Libc
  - Ret2syscall
  - One Gadget

CRAX++ Config

```
...
pluginsConfig.CRAX = {
    showInstructions = false,
    showSyscalls = true,
    concolicMode = true,

    modules = {
        "CodeSelection",   -- CRAX   (2014)
        "IOStates",        -- LAEG   (2021)
        "DynamicRop"       -- CRAX++ (2022)
    },

    techniques = {
        "Ret2csu",
        "AdvancedStackPivoting",
        "GotLeakLibc",
        "Ret2syscall"
    },
}
```

0x35 Extending CRAX++

- Modules (i.e. Plugins)
  - [LAEG] **I/O States**
  - [CRAX++] **Dynamic ROP**
  - [CRAX] **Code Selection**
  - ...

CRAX++ Config

```
...
pluginsConfig.CRAX = {
    showInstructions = false,
    showSyscalls = true,
    concolicMode = true,

    modules = {
        "CodeSelection",   -- CRAX   (2014)
        "IOStates",        -- LAEG   (2021)
        "DynamicRop"       -- CRAX++ (2022)
    },

    techniques = {
        "Ret2csu",
        "AdvancedStackPivoting",
        "GotLeakLibc",
        "Ret2syscall"
    },
}
```

0x35 Extending CRAX++

- Modules (i.e. Plugins)
  - [LAEG] **I/O States** - Generate information leak exploit scripts.
  - [CRAX++] **Dynamic ROP** - ROP inside S^2E as we add exploit constraints.
  - [CRAX] **Code Selection** - Reduce the complexity of path constraints.
  - ...

0x35 Ext

- Modules
  - i.e. “Plug
  - We can i into CRA

```
-- CRAX   (2014)
-- LAEG   (2021)
-- CRAX++ (2022)
```

Writing Your Own Module

For example, suppose we're going to create a module called "MyModule":

1. Create a directory named `MyModule` in **libs2eplugins/src/s2e/Plugins/CRAX/Modules/**.
2. In MyModule directory, create two files:
   - MyModule.h
   - MyModule.cpp

```
// libs2eplugins/src/s2e/Plugins/CRAX/Modules/MyModule/MyModule.h
#ifndef S2E_PLUGINS_CRAX_MY_MODULE_H
#define S2E_PLUGINS_CRAX_MY_MODULE_H

#include <s2e/Plugins/CRAX/Modules/Module.h>

namespace s2e::plugins::crax {

class MyModule : public Module {
public:
    class State : public ModuleState {
    public:
        State() : ModuleState() {}
        virtual ~State() override = default;

        static ModuleState *factory(Module *, CRAXState *) {
            return new State();
        }
```

0x35 Extending CRAX++

- A CRAX++ module has access to these API
  - **Instruction / Syscall hooks** - runtime instrumentation
  - **Memory and register** - read / write them as if you are automating gdb
  - **Virtual memory map** - a llvm::IntervalMap that works like pwndbg's vmmap
  - **Disassembler** - disassemble a list of raw bytes, or a given function
  - **VM snapshot** - unconditionally fork an execution state[1]

- You can also write a module and override the default exp. generator

[1] S2E/s2e PR#34 - klee,s2e: added support for fork() without symbolic conditions (URL: https://github.com/S2E/s2e/pull/34)

0x36 Summary

- CRAX++ (2022)
  - Written in C++17 (~8000 LoC), based on S^2E 2.0
  - Targets x86_64 Linux ELF
  - **Exploit Techniques**
    - Ret2stack, Ret2csu, Ret2syscall
    - StackPivoting * 2, GotLeakLibc, OneGadget [1]
  - **Modules (Plugins)**
    - I/O States [2], Dynamic ROP, Code Selection [3]

[1] david942j. “一發入魂 One Gadget RCE”. HITCON CMT 2017.
[2] W.-L. Mow, S.-K. Huang, H.-C. Hsiao “LAEG: Leak-based AEG using Dynamic Binary Analysis to Defeat ASLR.” The 6th International Workshop on Privacy, data Assurance, Security Solutions for Internet of Things, June 2022.
[3] Huang, Shih-Kun, et al. "Software crash analysis for automatic exploit generation on binary programs." IEEE Transactions on Reliability (2014).

Conclusion

Table CTF (Pwn) Binaries and CVE Binaries Successfully Exploited by CRAX++

Binary (x86_64)	Source / Advisory ID	Input Source	Vuln. Type	PoC Input Size (Bytes)	Exploit Gen. Time (sec.) Stage1 / Stage 2 / Total	ASLR	NX	PIE	Canary	Full RELRO
aslr-nx-pie-canary-fullrelro-trans	CRAXplusplus	stdin	Local Stack	1024	89 / 37 / 126	✓	✓	✓	✓	✓
aslr-nx-pie-canary-fullrelro	CRAXplusplus	stdin	Local Stack	1024	87 / 39 / 126	✓	✓	✓	✓	✓
aslr-nx-pie-canary	CRAXplusplus	stdin	Local Stack	1024	57 / 24 / 81	✓	✓	✓	✓	
aslr-nx-pie	CRAXplusplus	stdin	Local Stack	345	82 / 31 / 113	✓	✓	✓		
aslr-nx-canary	CRAXplusplus	stdin	Local Stack	345	53 / 32 / 85	✓	✓		✓	
aslr-nx	CRAXplusplus	stdin	Local Stack	1024	11 / - / 11	✓	✓			
speedrun-002	DEFCON'27 CTF Quals	stdin	Local Stack	2247	14 / - / 14	✓	✓			
no_canary	angstromctf 2020	stdin	Local Stack	208	157 / - / 157	✓	✓			
tranquil	angstromctf 2021	stdin	Local Stack	512	28 / - / 28	✓	✓			
bof: 5 pt	pwnable.kr	stdin	Local Stack	512	28 / - / 28	✓	✓			
unexploitable: 500 pt	pwnable.kr	stdin	Local Stack	512	13 / - / 13	✓	✓			
unexploitable: 500 pts	pwnable.tw	stdin	Local Stack	1024	15 / - / 15	✓	✓			
unexploitable-trans	CRAXplusplus	stdin	Local Stack	1024	16 / - / 16	✓	✓			
ret2win	ROP Emporium	stdin	Local Stack	512	12 / - / 12	✓	✓			
split	ROP Emporium	stdin	Local Stack	512	11 / - / 11	✓	✓			
callme	ROP Emporium	stdin	Local Stack	512	13 / - / 13	✓	✓			
readme	NTU Computer Security 2017	stdin	Local Stack	1024	15 / - / 15	✓	✓			
readme-alt1	CRAXplusplus	stdin	Local Stack	1024	14 / - / 14	✓	✓			
readme-alt2	CRAXplusplus	stdin	Local Stack	1024	14 / - / 14	✓	✓			
dnsmasq (2.77)	CVE-2017-14993	socket	Remote Stack	1574	105 / - / 126	✓	✓			
dnsmasq (2.77)	CVE-2017-14993	socket	Remote Stack	238	112 / - / 113					
rsync (2.5.7)	CVE-2004-2093	env	Local Stack	141	33 / - / 33					
ncompress (4.2.4)	CVE-2001-1413	arg	Local Stack	1054	69 / - / 69					
glftpd (1.24)	OSVDB-ID-16373	arg	Local Stack	286	30 / - / 30					
iwconfig (v26)	BID-8901	arg	Local Stack	94	28 / - / 28					

CTF

CVE

0x41 Results

- Real-World Targets
  - CVE-2017-14993 dnsmasq (2.77)
  - CVE-2004-2093 rsync (2.5.7)
  - CVE-2001-1413 ncompress (4.2.4)
  - OSVDB-ID-16373 glftpd (1.24)
  - BID-8901 iwconfig (v26)
- CTF Binaries
  - DEFCON'27 CTF Quals - speedrun002
  - pwnable.kr unexploitable (500 pt)
  - pwnable.tw unexploitable (500 pts)
  - angstromctf 2020 no_canary
  - angstromctf 2021 tranquil
  - aslr-nx-pie-canary-fullrelro
  - aslr-nx-pie-canary
  - aslr-nx-pie, aslr-nx-canary
  - aslr-nx
  - ...

0x41 **Results**

- NTU Computer Security 2017: Readme (150 pts) Revenge
    - ASLR + NX
    - We can only overwrite
        - saved RBP
        - return address
    - **Pwned**

```c
3 int main() {
4     char buf[0x20];
5     read(0, buf, 0x30);
6 }
```

0x41 Results

- pwnable.tw: Unexploitable (500 pts) Revenge
  - ASLR + NX
  - No `syscall` gadget
  - Your payload will be reversed
  - **Pwned**

```
8  int main() {
9      sleep(3);
10     char buf[4];
11     read(0, buf, 0x100);
12     std::reverse(buf, buf + 0x100);
13 }
```

0x41 **Results**

- aslr-nx-pie-canary-fullrelro
  - All protections enabled
    - Except FORTIFY
  - Information Leak
    - **2 chances**
  - We can overwrite
    - canary
    - saved RBP
    - return address
  - **Pwned**

```c
4  int main() {
5      setvbuf(stdin, NULL, _IONBF, 0);
6      setvbuf(stdout, NULL, _IONBF, 0);
7
8      char buf[0x18];
9      printf("what's your name: ");
10     read(0, buf, 0x80);
11
12     printf("Hello, %s. Your comment: ", buf);
13     read(0, buf, 0x80);
14
15     printf("Thanks! We've received it: %s\n", buf);
16     read(0, buf, 0x30);
17 }
```

0x41 Results

- CVE-2017-14493 dnsmasq
  - ASLR + NX
  - Stack-buffer overflow via a crafted DHCPv6 request
  - Exploitation
    - Grab a PoC from exploit-db, and write the crafted DHCPv6 packet to a file.
    - CRAX++ can turn that **PoC DHCPv6 packet** into an **ROP exploit script** for you.

0x42 Future Work

- Stack pivoting multiple times
  - currently CRAX++ can only pivot the stack once (to .bss)
- Enhance Dynamic ROP
  - partially overwrite return address
- Symbolic pointers
  - enables CRAX++/S2E to solve more complicated path constraints
- Automated heap exploitation
  - explore not only the crashing path, but also diverging paths

Thanks

- Prof. Shih-Kun Huang of SQLab, NYCU
- Prof. Hsu-Chun Hsiao of NSLab, NTU
- @how2hack of Balsn CTF Team
- All developers of the original CRAX

Thanks for your time!

aesophor