Exploitation Toolkit: Icarus

George Nicolaou
Glafkos Charalambous

Objectives

- Reveal the architecture behind the Icarus Library
- Discuss the methodologies used in some of the modules
- Release beta version of the toolkit (Finally!)
- Get your support

Outline

- Toolkit Architecture
  - Icarus Disassembly Engine (iDisasm)
  - ETI Library
- Toolkit Modules
  - Instruction Finder
  - Exploitability Analysis
  - Gadget Finder
- Contribution and Road Ahead

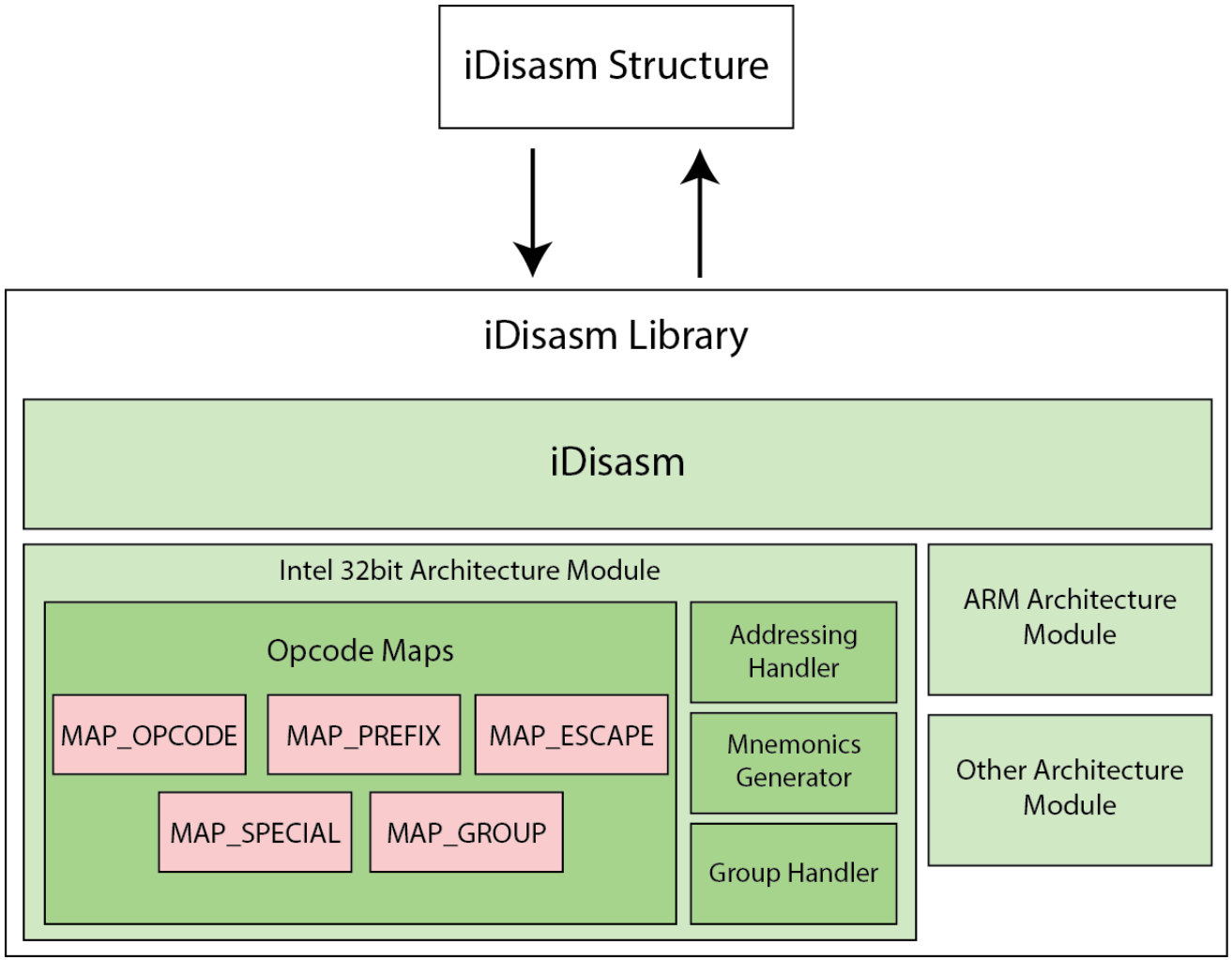
Toolkit Architecture

Icarus Disassembly Engine (iDisasm)

iDisasm

- Yet another disassembler, why?
  - Cross platform supported architecture.
  - Easy maintenance & understanding.
  - No wild dependencies in Icarus Engine.
- Architecture
  - A Single structure containing information about the instruction.
  - Everything is stored in arrays.

iDisasm


iDisasm Structure
iDisasm Library
iDisasm
Intel 32bit Architecture Module
Opcode Maps
MAP_OPCODE
MAP_PREFIX
MAP_ESCAPE
MAP_SPECIAL
MAP_GROUP
Addressing Handler
Mnemonics Generator
Group Handler
ARM Architecture Module
Other Architecture Module

iDisasm

```
typedef struct {
    ARCHITECTURE Architecture;
    uiptr InstructionPointer;
    uiptr VirtualAddress;
    INSTRUCTION Instruction;
    int32 InstructionSize;
    PREFIXES Prefixes;
    char Mnemonic[MAX_INSTRUCTION_MNEMONIC_SIZE];
    struct {
        int nOpcodeIndex;
        int nVADelta;
        uiptr OldVirtualAddress;
        BOOL bPopulateMnemonics;
    } InternalStuff;
} SIDISASM, * PSIDISASM;
```

- Architecture
  - X86, X86_64 ARM
- Instruction Pointer
  - Location of the instructions buffer
- VirtualAddress
  - Actual address in memory
- Instruction
  - Information about the instruction
- Prefixes
  - Prefix Flags
- Mnemonic
  - Instruction mnemonic

iDisasm

```
typedef struct {
    ENUM_OPCODE_CATEGORY_T Category;
    OPERAND Operands[3];
    int32 Opcode;
    REG ModifiedRegisters;
    BOOL bIsValid;
    char InstructionMnemonicName[26];
} INSTRUCTION;
```

- Category
  - Instruction category (CONTROL_TRANSFER, ARITHMETICAL, etc)
- Opcode
  - Instruction's opcode
- ModifiedRegs
  - Registers that are modified in this instruction
- InstructionMnemonicName
  - Instruction name (MOV, CMP, ADD, JMP)

iDisasm

```
typedef struct {
    OPERAND_TYPE Type;
    OPERAND_ACCESS Access;
    BITSIZE BitSize;
    BITSIZE AddressingSize;
    ADDRESSING_REGISTER AddrRegister;
    union {
        REG Register;
        uiptr Value;
        int32 RelAddress;
        struct {
            REG BaseRegister;
            REG IndexRegister;
            int8 Scale;
            int32 Displacement;
        } Memory;
    } RegValMem;
    uiptr TargetAddress;
} OPERAND;
```

- Type
  - High Byte contains the operand type
    - OPERAND_REGISTER
    - OPERAND_MEMORY
    - Etc
  - Low 3 bytes contain register type (overwritten by architecture)
    - TYPE_REG_GENERAL
    - TYPE_REG_SPECIAL_0
    - etc

iDisasm

```
typedef struct {
    OPERAND_TYPE Type;
    OPERAND_ACCESS Access;
    BITSIZE BitSize;
    BITSIZE AddressingSize;
    ADDRESSING_REGISTER AddrRegister;
    union {
        REG Register;
        uiptr Value;
        int32 RelAddress;
        struct {
            REG BaseRegister;
            REG IndexRegister;
            int8 Scale;
            int32 Displacement;
        } Memory;
    } RegValMem;
    uiptr TargetAddress;
} OPERAND;
```

- Access
  - READ/WRITE/READWRITE
- BitSize
  - The size of the operand
- AddressingSize
  - The addressing size (when memory)
- AddrRegister
  - Addressing register (DS,FS)
- RegValMem
  - Contains the register/value or memory information of the operand

iDisasm

- iDisasm library exports a single function
  - int disasm(PSDISASM arg);
- Loads the architecture module specified by
  - arg->Architecture
- Loads the first *Opcode Map*, recurses through it and generates information about the instruction

```
const  PLUGIN_MAP one_byte_opcode_map[] = {
    /* 00 - 07 */
    { MAP_OPCODE, { OP_ADD,     {{ ADDR_E, TYPE_b, WRITE }, { ADDR_G, TYPE_b, READ }} } },
    { MAP_OPCODE, { OP_ADD,     {{ ADDR_E, TYPE_v, WRITE }, { ADDR_G, TYPE_v, READ }} } },
    { MAP_OPCODE, { OP_ADD,     {{ ADDR_G, TYPE_b, WRITE }, { ADDR_E, TYPE_b, READ }} } },
    { MAP_OPCODE, { OP_ADD,     {{ ADDR_G, TYPE_v, WRITE }, { ADDR_E, TYPE_v, READ }} } },
```

iDisasm

- New modules can be defined by populating the *ArchitectureModules[]* array in *archs.h*

```
typedef struct _architecture_module {
    ARCHITECTURE Architecture;
    PLUGIN_MAP * lpInitialPluginMap;
    ENUM_OPCODE_CATEGORY_T ( * get_opcode_category )( ENUM_OPCODE_PTR );
    int ( * addressing_handler )( PSIDISASM, INSTRUCTIONINTERNAL * );
    int ( * group_handler )( PSIDISASM, GROUP_MAP * );
    BOOL ( * get_mnemonic)( PSIDISASM );
} ARCHITECTURE_MODULE;


ARCHITECTURE_MODULE ArchitectureModules[] = {
    {
        X86,
        (PLUGIN_MAP *)&one_byte_opcode_map,
        &intel_get_opcode_category,
        &intel_addressing_handler,
        &intel_group_handler,
        &intel_get_mnemonic
    },
    { ARM, NULL }
};
```

iDisasm

- Conclusions
  - iDisasm is a crucial module in successfully porting the Icarus engine in multiple processor platforms.
  - Next targeted platform is ARMv7
- License
  - GNU Lesser General Public License v3 (LGPLv3)

Toolkit Architecture

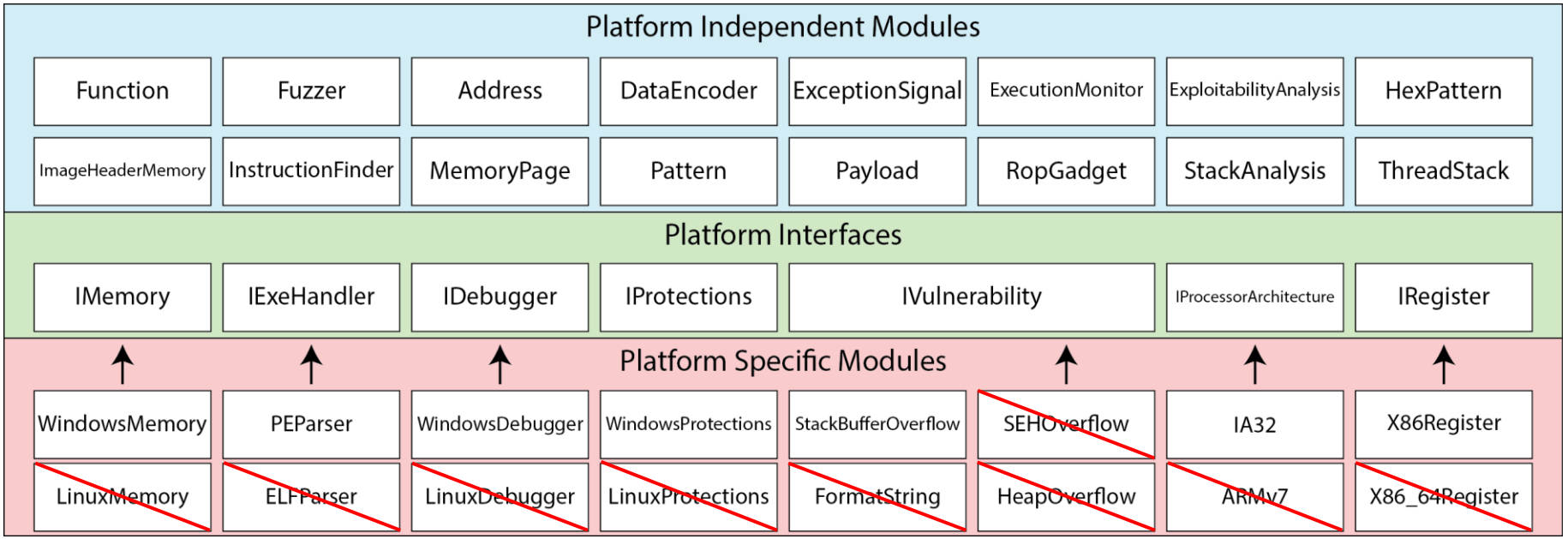
Exploitation Toolkit Icarus Library

ETI Library

- The ETI Library exposes a set of modules/tools for assisting the development of POC exploit code.
- Architecture Layers
  - Platform Specific Modules
    - Debugger
    - Executable file handlers
    - Etc
  - Platform Interfaces
    - Various Interfaces (C++)
    - Abstract structures and general algorithms
    - Linking various platforms together
  - Platform Independent Modules
    - Variety of modules using *Platform Interfaces*

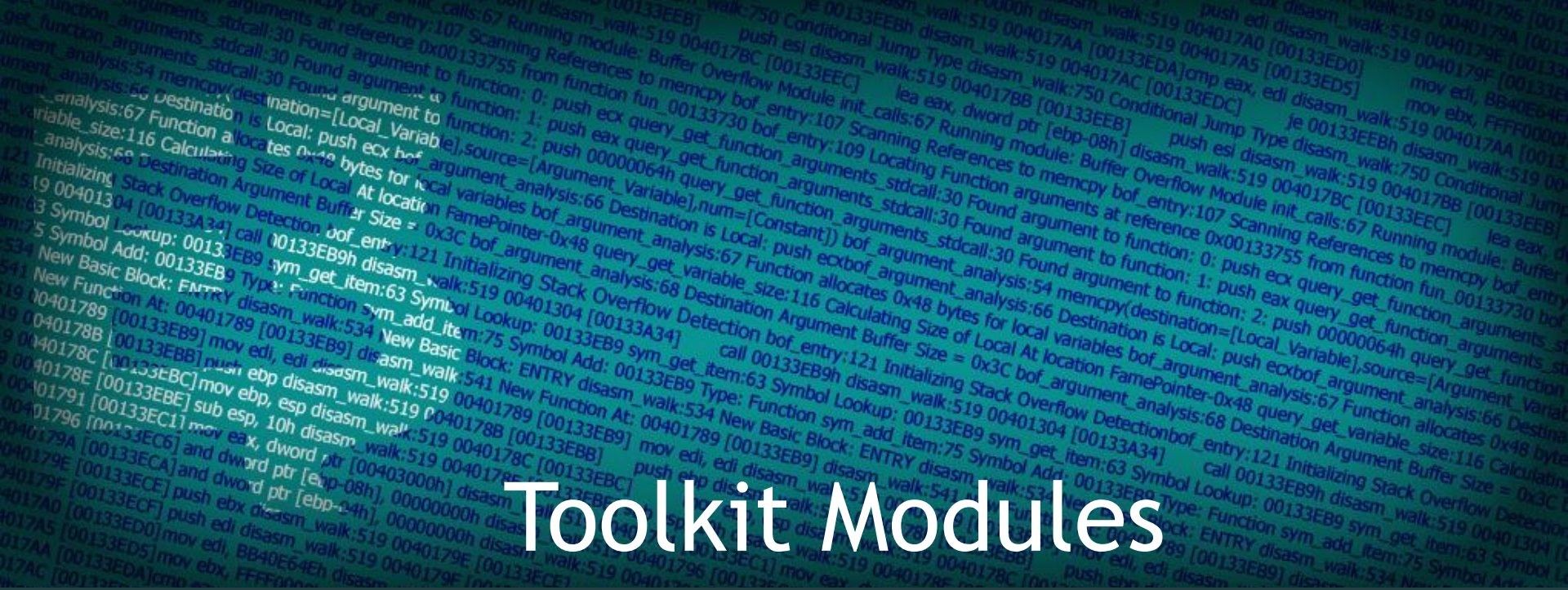
ETI Library

- Main modules listed for each layer
- Crossed modules are not implemented yet
- Minimal coding requirements for new platforms



ETI Library

- License
  - GNU General Public License v3
  - Planning to switch to Lesser at a much later point



Toolkit Modules

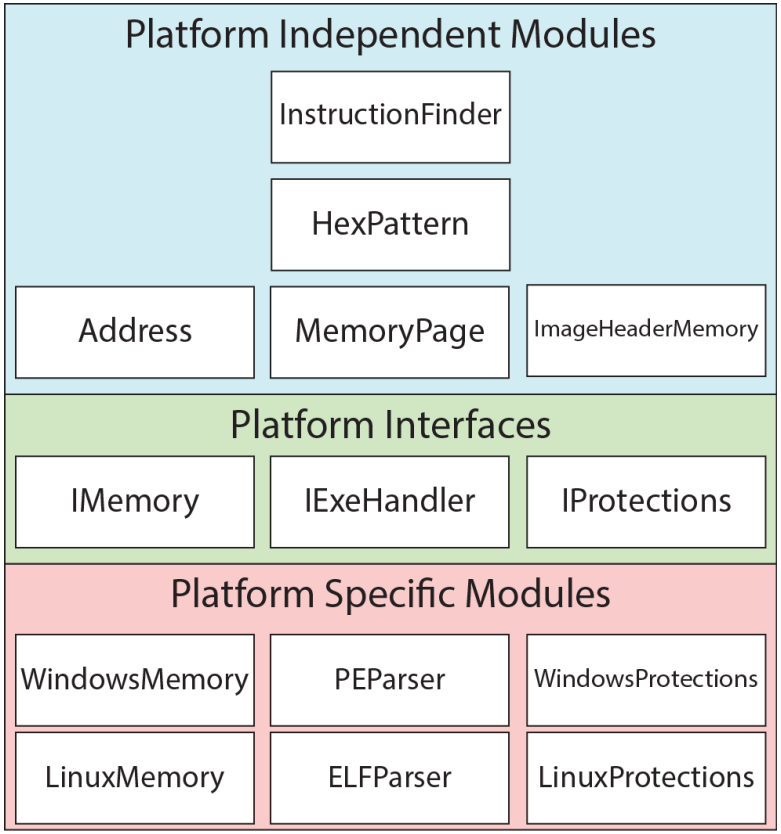
Toolkit Modules

- How modules fit into the architecture
  - Pattern Creator
    - Typical cyclic pattern generation
    - Multiple sets support
  - Instruction Finder
    - Cross platform design
    - Locating instructions in executable memory pages given a hexadecimal pattern
  - Exploitability Analysis
    - Analyse and report vulnerability information right after an exception occurs
  - Gadget Finder
    - Locate ROP gadgets in memory
    - Parse and populate attribute information for each gadget

Toolkit Modules

Instruction Finder

Instruction Finder

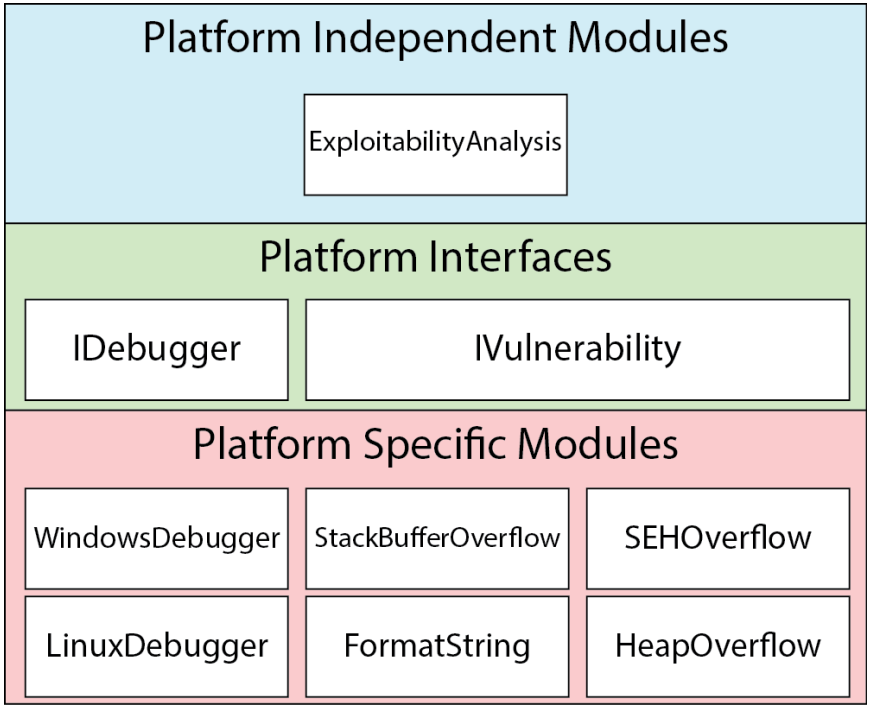


- Generalized architecture
  - Making use of *IMemory* to retrieve pages and module headers
  - Parsing headers using *IExeHandler*
  - Filtering headers using *IProtections*
- InstructionFinder exposes the function *find_instruction_in_exe* that receives
  - Int – Process Id.
  - IProtections – Protections Filter
  - HexPattern – Compiled Hex Pattern
  - vector<Address *> - Vector that receives found addresses

Toolkit Modules

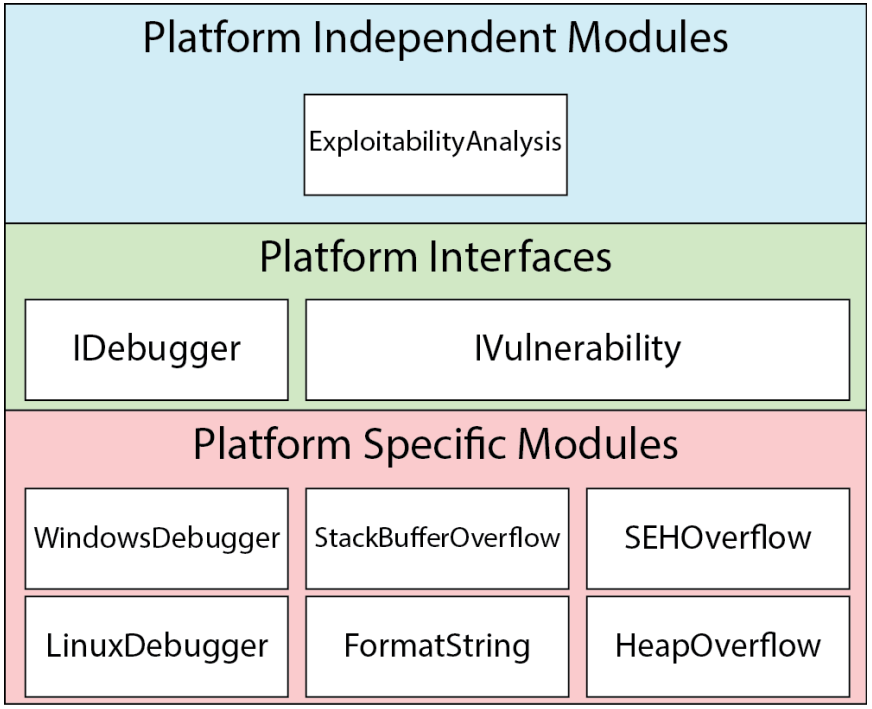
Exploitability Analysis

Exploitability Analysis



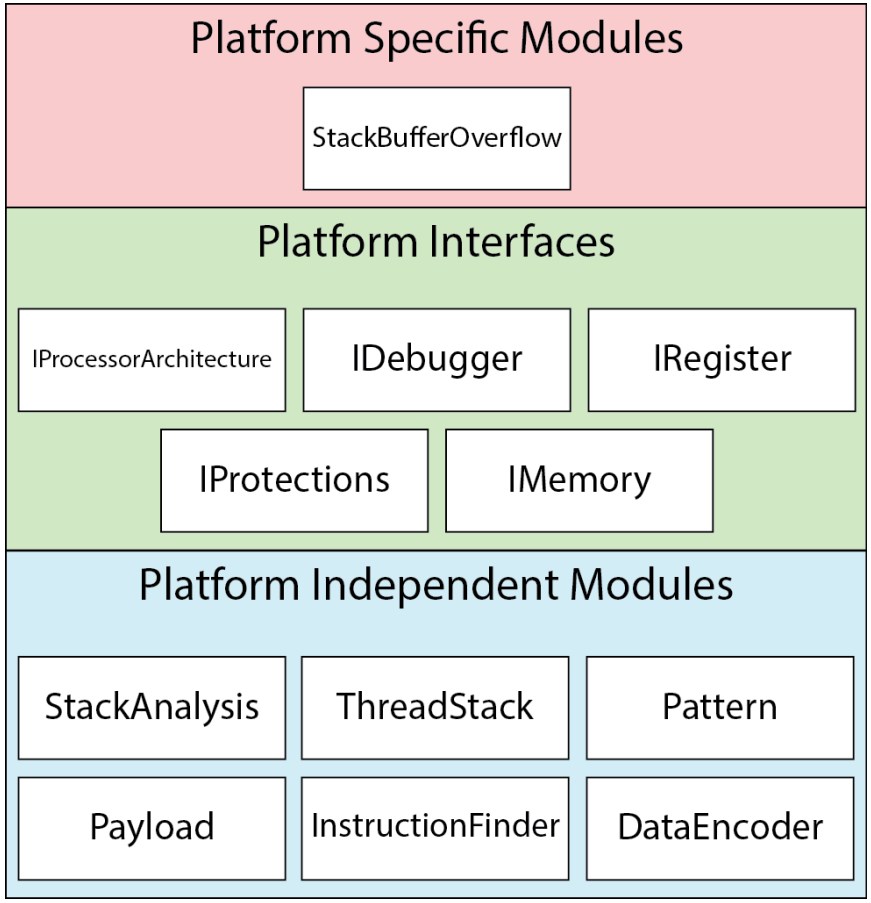
- An “interface” between the front end and back end
- Provides
  - Vulnerability Classification
  - Vulnerability Analysis & Implementation
- Architecture
  - Debugs target application
  - Runs specified vulnerability modules for providing the above services.

Exploitability Analysis



- IVulnerability Modules expose 3 functions
  - check_for_vulnerability()
  - run_vulnerability_analysis()
  - run_skeleton_implementation()
- Each function "scores" the vulnerability on how it can be (ab)used if possible
- Analysis & Implementation parts generate a *Payload* object (one more complete than the other)

IVulnerability: StackBufferOverflow



- check_for_vulnerability()
  - Pull the registers of the excepting thread and check common patterns against them.
  - Store any registers that might be controllable.
- run_vulnerability_analysis()
  - Run controllable register lookups
  - Locate overwrite offsets.
  - Use of a cyclic pattern is not required.
- run_skeleton_implementation()
  - Find corrupted bytes.
  - Locate valid return addresses based on IProtections filters.
  - Produce a working *Payload* object

Use of a cyclic pattern is not required?

- Well yes, assuming that
  - If the overflow buffer is filled with AAAs or BBBs
  - And the last RETN instruction did not consume any bytes (eg RETN 4)
- The technique is relatively simple (assume IA32)
  - Locate EIP relative to ESP (at location **ESP-N**)
  - Check if:

$$ESP - N - PATTERN_{WRAPSIZE} + PATOFFSET_{EIP} == PATTER_{LAST4BYTES}$$

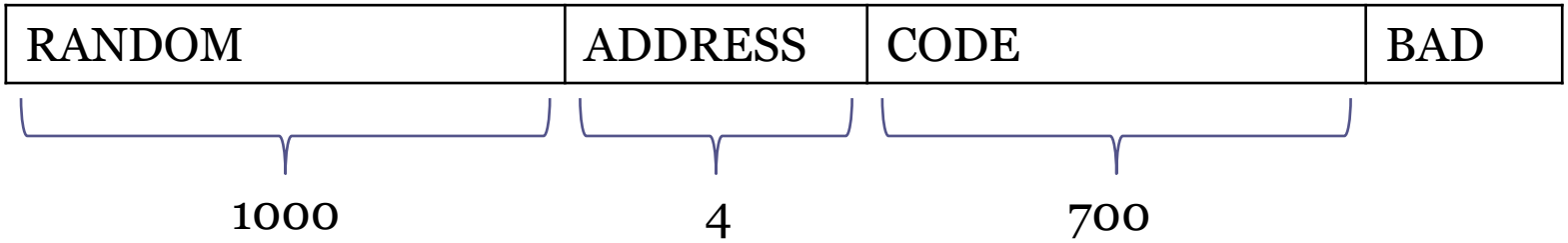
  - If equal then loop going backwards PATTERN_WRAPSIZE else we got a matching offset
  - Works best with a cyclic pattern

Payload Object

- A Payload object contains a linked list that describes the payload structure.
- Each element in the list has
  - A type
    - PAYLOAD_RANDOM
    - PAYLOAD_ADDRESS
    - PAYLOAD_ADDRESS_MULTIPLE
    - PAYLOAD_CODE
    - PAYLOAD_BAD
    - PAYLOAD_FIXED
  - Size
  - Contents
  - A set of restricted characters

Payload Object

- Using the Payload object you can construct something like



- The address element on top can contain all possible addresses that suit your requirements

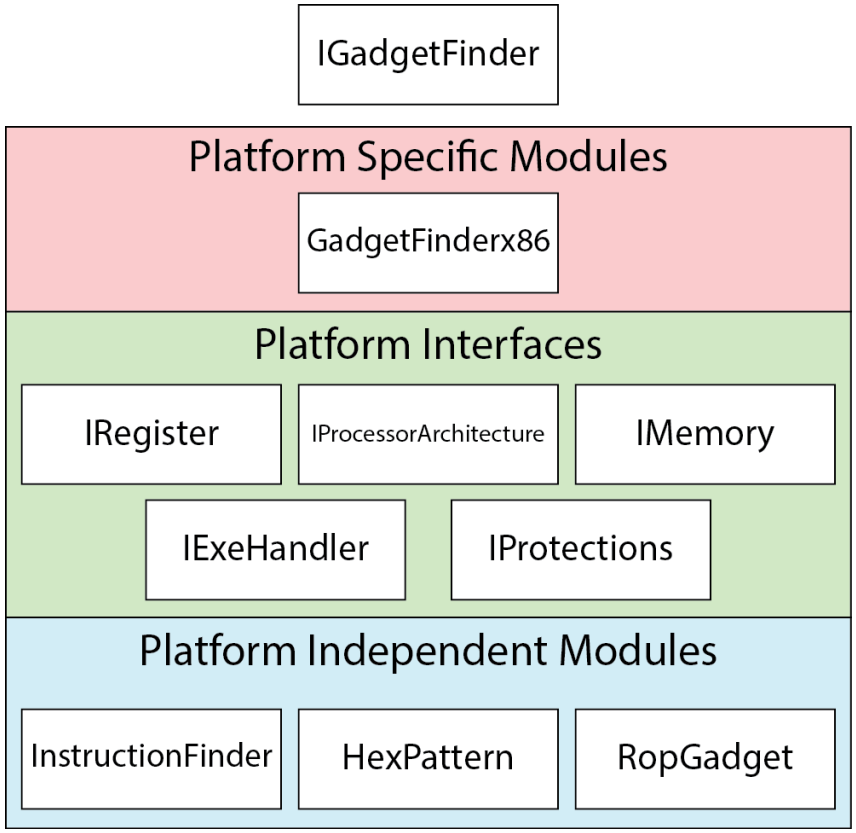
Exploitability Analysis

Demo

Toolkit Modules

Gadget Finder

Gadget Finder



- Gadget finder is responsible for locating ROP Gadgets in executable memory addresses
- Two main functions exposed so far
  - proc_find_rop_gadgets()
  - proc_find_api_gadgets()
- Resulting gadgets are stored in a vector containing RopGadget objects
- Callers can retrieve gadgets by using
  - get_found_rop_gadgets()
  - get_found_api_gadgets()

Gadget Finder

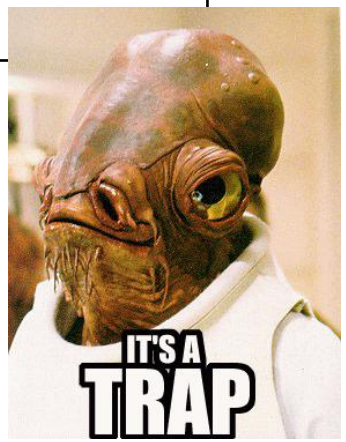
- Gadgets are divided into two main entities
  - **API Gadgets –** Gadgets calling library functions
  - **Standard ROP Gadgets** – Your standard do something then return gadgets
- Locating API Gadgets
  - Apply protection filters
  - Look for JMP [IAT Address] instructions (FF 25)
  - Match address with an exported library function

Gadget Finder

- Locating Standard gadgets
  - Assume the following code:

0x10001020	55	PUSH EBP
0x10001021	8B EC	MOV EBP, ESP
0x10001023	8B 45 08	MOV EAX, DWORD SS:[EBP+8]
0x10001026	89 41 04	MOV DWORD DS:[ECX+4], EAX
0x10001029	5D	POP EBP
0x1000102A	C2 04 00	RETN 4

  - How many possible usable gadgets can you identify?



Gadget Finder

0x10001020	55	PUSH EBP
0x10001021	8B	MOV EBP, ESP
0x10001022	EC	
0x10001023	8B	MOV EAX, DWORD SS:[EBP+8]
0x10001024	45	
0x10001025	08	
0x10001026	89	MOV DWORD DS:[ECX+4], EAX
0x10001027	41	INC ECX
0x10001028	04	ADD AL, 5D
0x10001029	5D	POP EBP
0x1000102A	C2	RETN 4
0x1000102B	04	
0x1000102C	00	

Gadget Finder

- Locating Standard Gadgets
  1. Apply protection filters
  2. Look for RETN instructions (C3, C2, CA, CB)
  3. Walk backwards 1 byte at a time and disassemble instruction (byte-by-byte lookup)
     - Verify that instruction is valid.
     - Verify that new instruction doesn't overwrite the RETN instruction.
  4. Repeat for MaxRopSize instructions
  5. Attribute RopGadget (Affected regs, category, type, etc)

Gadget Finder

- Each gadget is attributed a *Category* as follows

Category	Description
GC_MEMORY	References a memory location
GC_REGMEMORY	References a memory location relative to a register
GC_ASSIGNMENT	Assigns a value to an operand
GC_SYSCALL	Contains a SYSCALL instruction
GC_MATH	Contains mathematical computations
GC_LOGICAL	Contains logical computations
GC_CONTROLFLOW	Contains control flow instructions
GC_SYSTEMINSTR	Contains a privileged instruction
GC_SEGMENT	References a segment register

Gadget Finder

- Each gadget is attributed a *Type* as follows

Type	Description
GT_CONTROLFLOW_REG	Control flow instruction references register
GT_CONTROLFLOW_MEM	Control flow instruction references memory
GT_CONTROLFLOW_REL	Control flow instruction is relative branch
GT_ASSIGNS_ZERO	Contains instruction that assigns zero to a register
GT_STRING_MOVE	Contains string move operation instruction
GT_STRING_CMP	Contains string comparison operation instruction
…	More to come

Gadget Finder

- Locating gadgets using the byte-by-byte lookup technique results into more usable gadget types
- Using the information populated in each RopGadget you can essentially
  - Locate specific gadgets for specific operations (even programmatically).
  - Automate the process of ROP Payload generation.

Gadget Finder

Demo

Contribution and Road Ahead

- Contributing to the Icarus Project
  - Although at early stages ETI can grow to become a valuable tool for exploit developers
  - GitHub (https://github.com/georgenicolaou/icarus)
  - Discussion forums (soon)
  - Homepage (soon)

Contribution and Road Ahead

- Who do we need
  - **Cross Platform UI Designers** (Qt looks nice)
  - **Android Developers**
  - Various Platform Coders (Linux/UNIX)
  - Beta testers
  - People that can pitch us ideas, requests and **beers**

Contribution and Road Ahead

- Road Ahead
  - iDisasm Support for multiple architectures
  - Complete Vulnerability Modules
    - Handle all possible cases and automate the process of exploitation
  - Spice up ExploitabilityAnalysis module
    - Execution Tracing
    - Locating vulnerable functions after trigger
  - Expand on GadgetFinder options
  - Build-in Fuzzer
  - Port to Linux and Android


Questions?