Bytecode Jiu-Jitsu

Choking Interpreters to Force Execution of Malicious Bytecode

Toshinori Usui[1], Yuto Otsuki[1]

Contributors: Ryo Kubota[1], Yuhei Kawakoya[1], Makoto Iwamura[1], Kanta Matsuura[2]

[1] NTT Security Holdings Corporation

[2] Institute of Industrial Science, The University of Tokyo

NTT Security Holdings

Toshinori Usui, Ph.D.

- Research scientist, security principal
- Research interests: malware analysis, reverse engineering, and exploit development
- CTF lover
- Brazilian Jiu-Jitsu enthusiast

Yuto Otsuki, Ph.D.

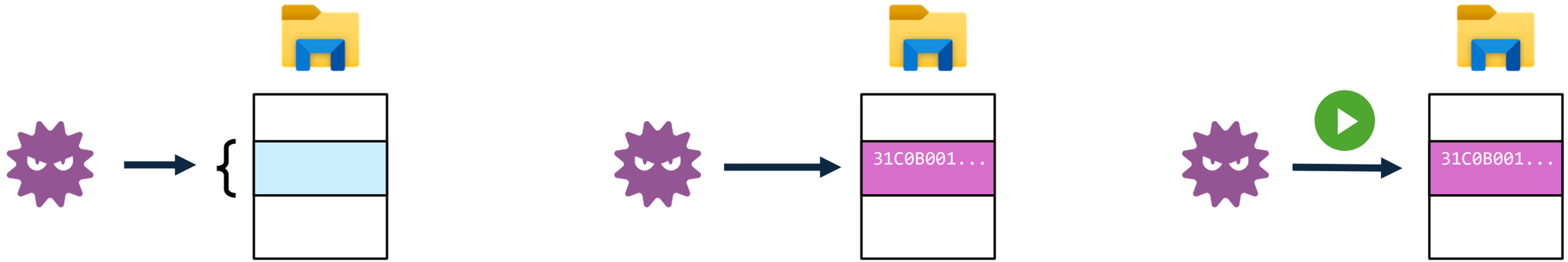
- Senior researcher
- Research interests: memory analysis, reverse engineering and operating system security

Code Injection Attack

1. Allocate
a memory region

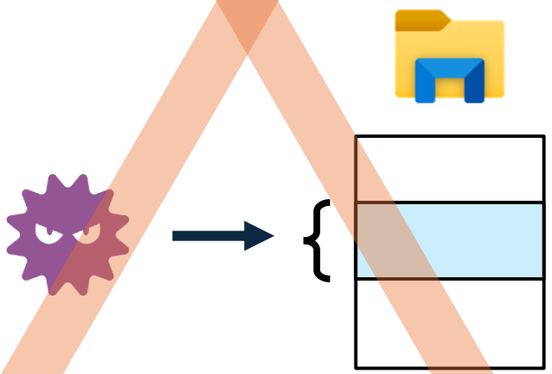
2. Write
malicious code

3. Execute
the code

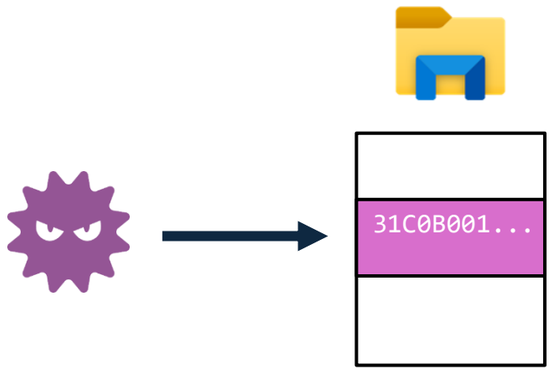


Code Injection Attack

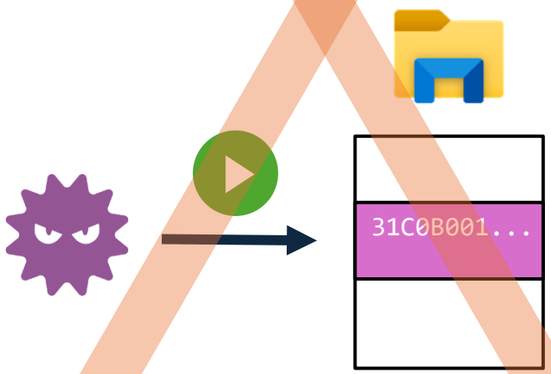
1. Allocate
   a memory region



2. Write
   malicious code

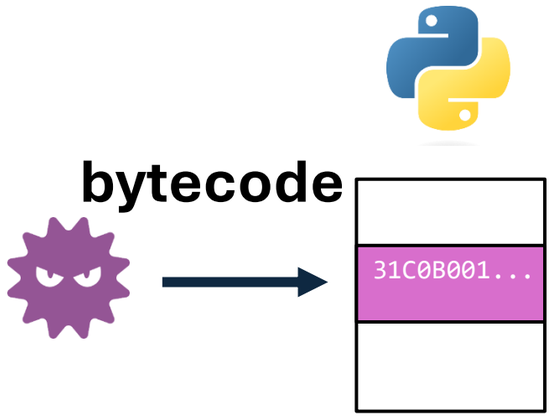


3. Execute
   the code



Code Injection Attack

2. Write
malicious code



Today's Topic: Bytecode Jiu-Jitsu

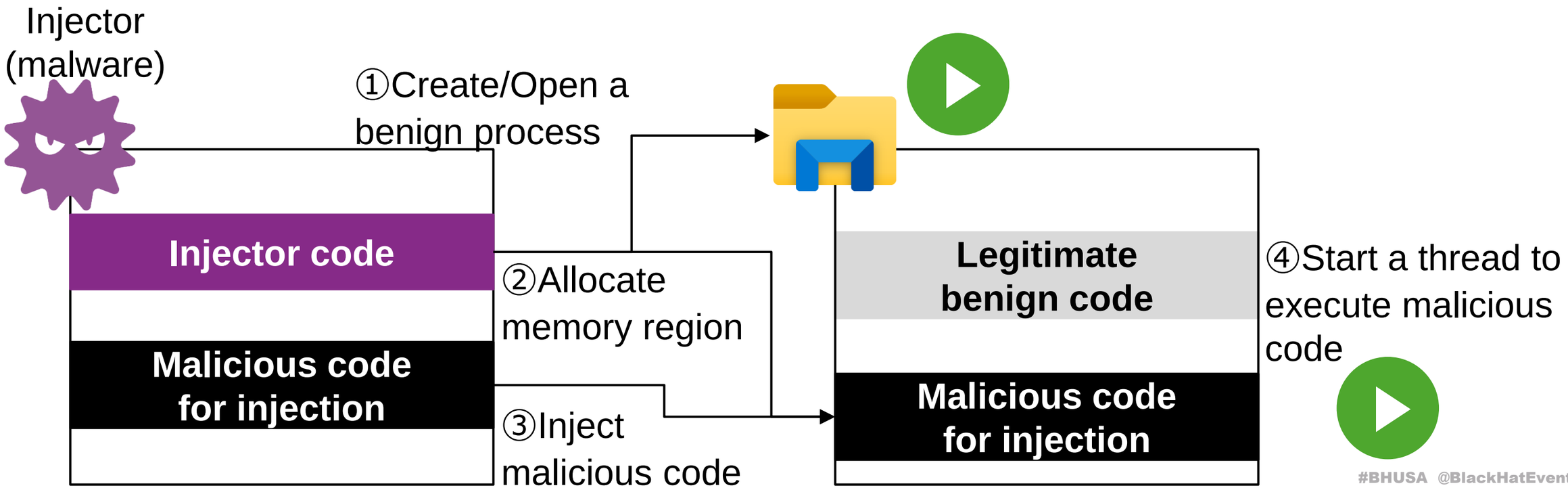
Outline

- 入門 Introduction to Code Injection Attack
- 理合 Bytecode Jiu-Jitsu Overview
- 稽古 Interpreter Implementation Basics
- 打込 Interpreter Analysis
- 試合 Bytecode Jiu-Jitsu Attack
- 乱取 Experiments and Evaluations
- 受身 Countermeasures against Bytecode Jiu-Jitsu
- 総括 Takeaways

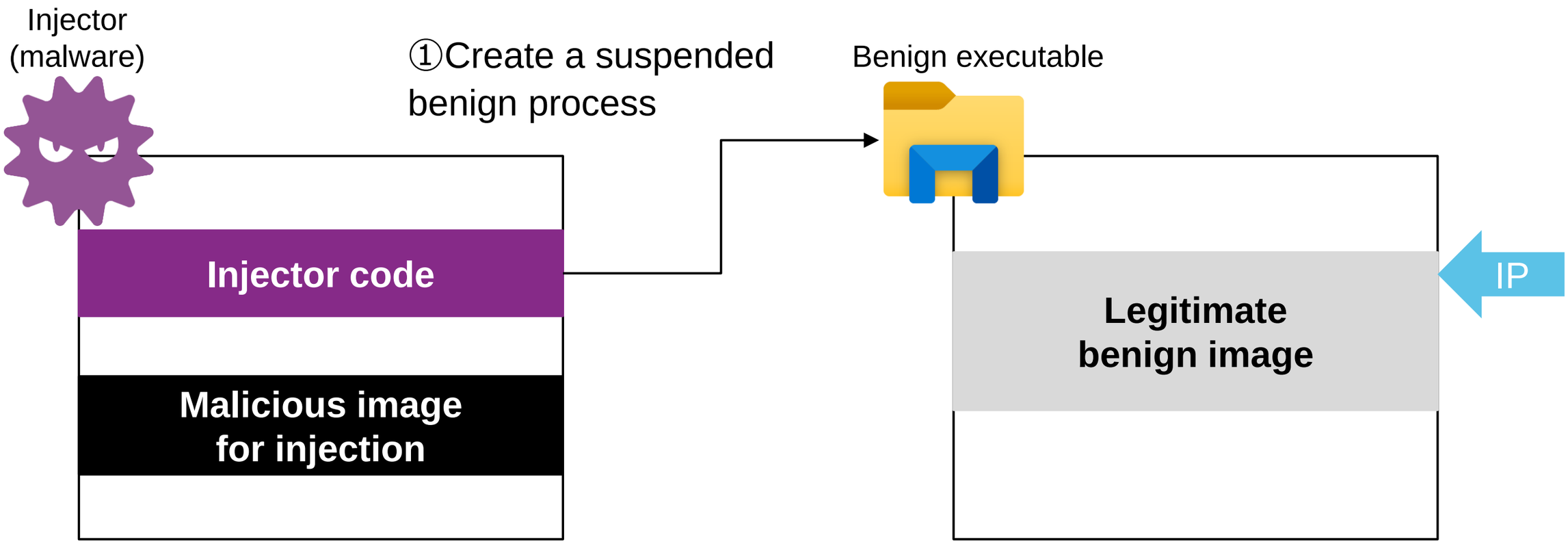
入門 Introduction to Code Injection Attack

Code Injection Attack

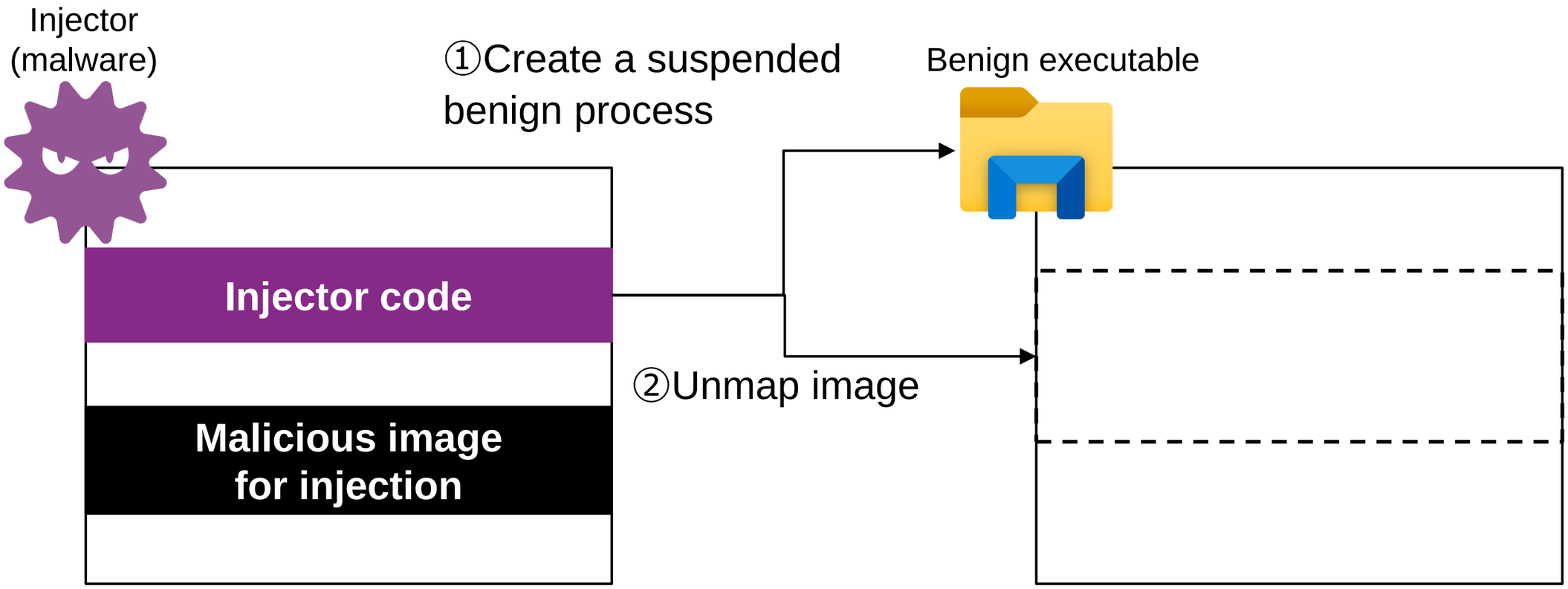
- Malware tries to conceal their malicious behavior on the target host
- Code injection is a technique to blend malicious behavior with benign one by forcing a benign process to execute malicious code



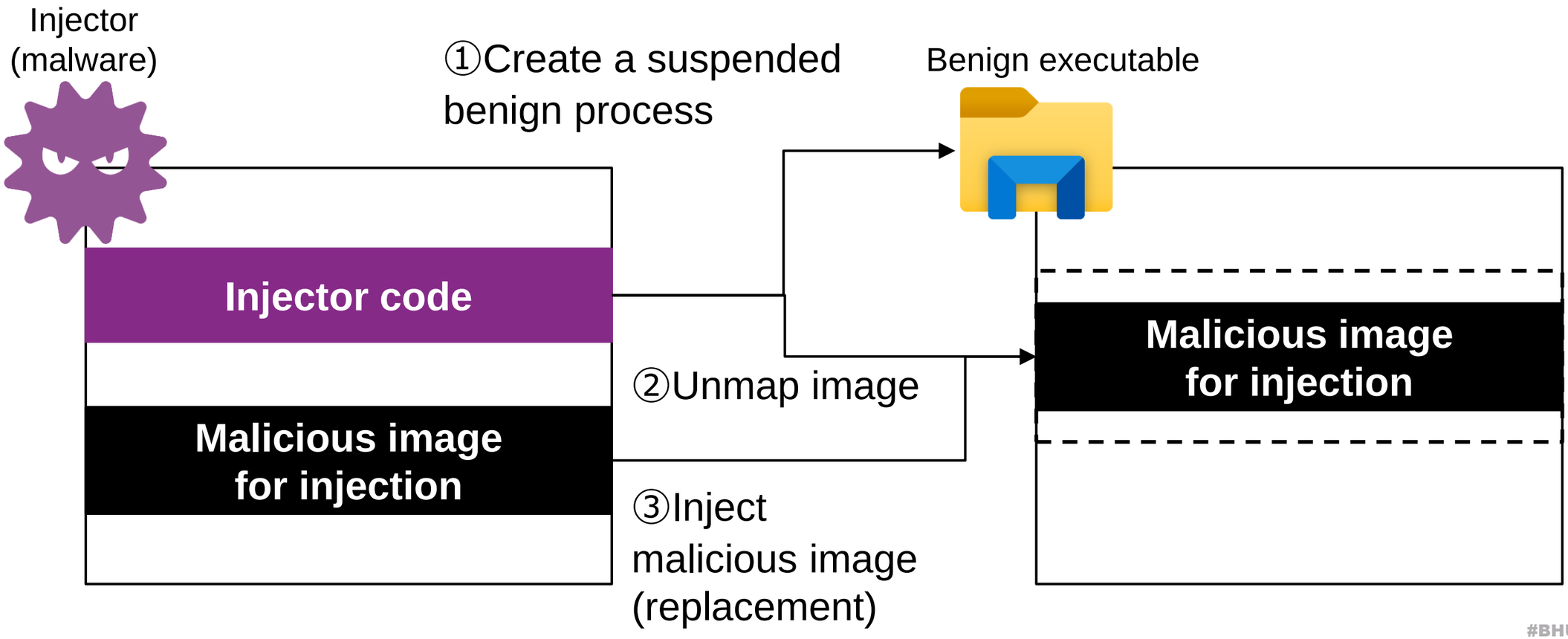
Process Hollowing



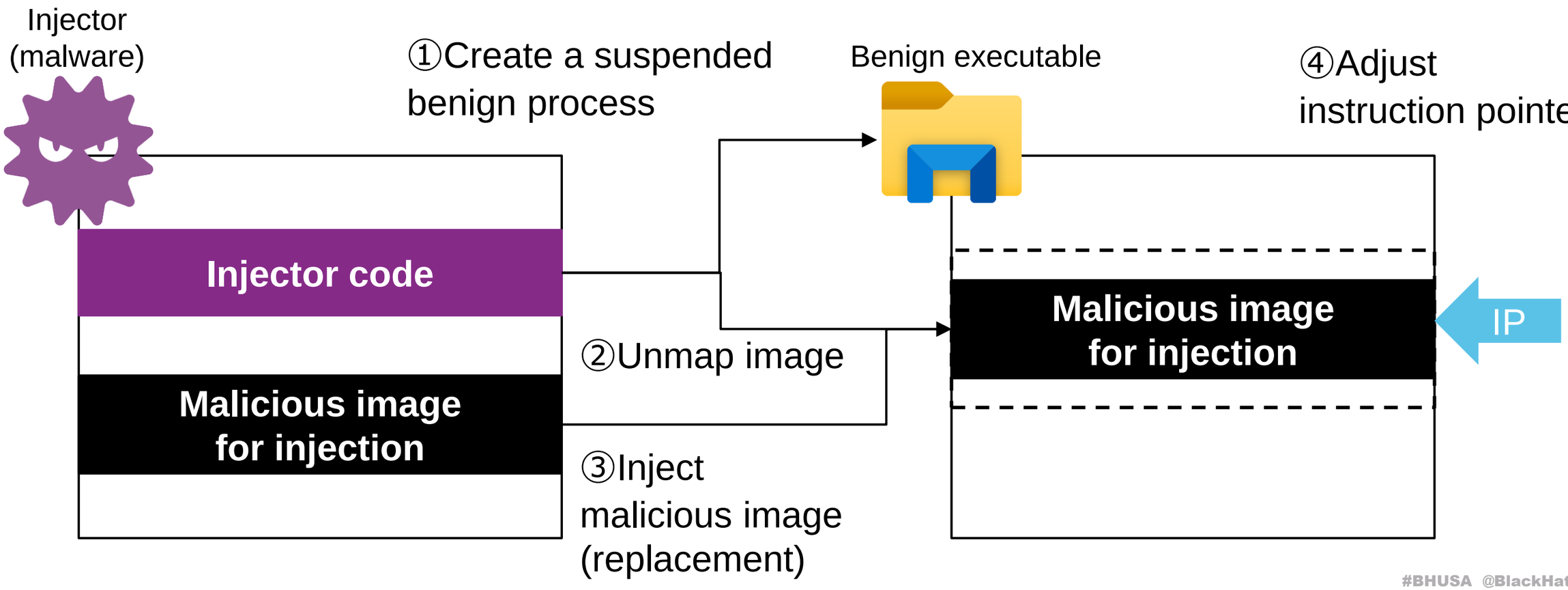
Process Hollowing



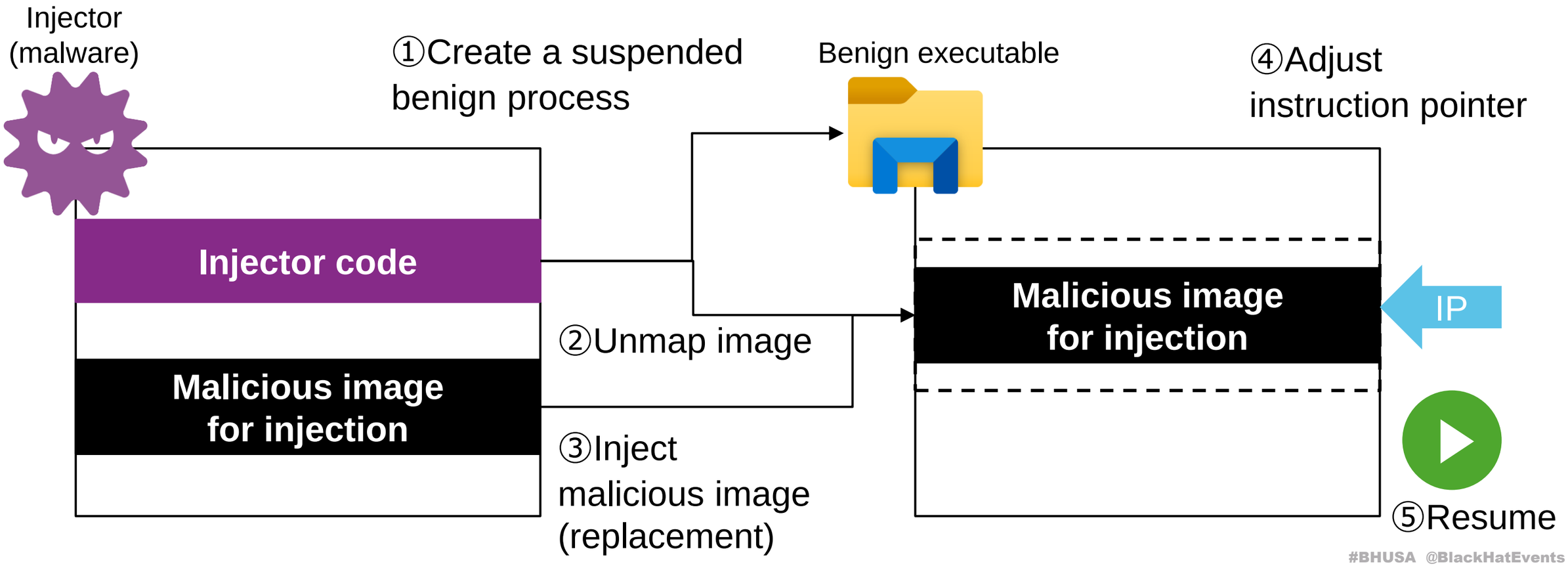
Process Hollowing



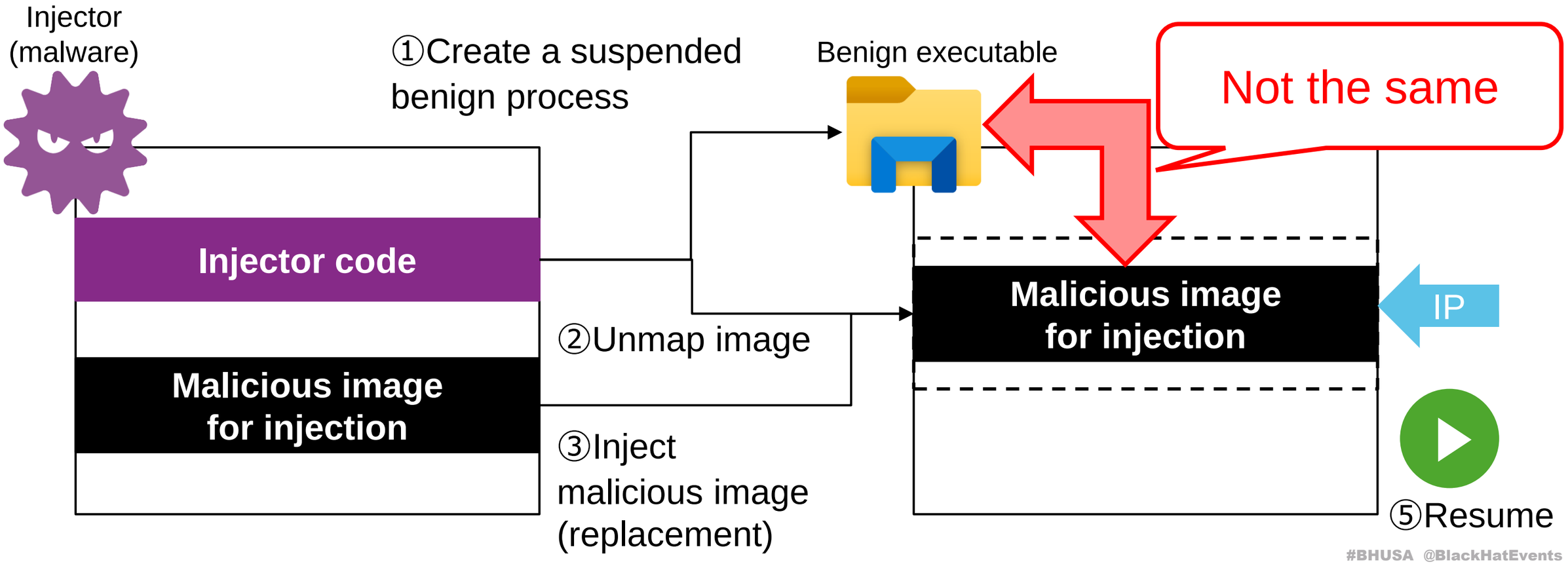
Process Hollowing



Process Hollowing



Process Hollowing



Process Hollowing Variants

- **Process Doppelgänging**
  1. Start a transaction and writes malicious code to a benign file
  2. Creates an in-memory image from the file
  3. **Rolls the file back**
  4. Creates a process from the image
- **Process Herpaderping**
  1. Writes malicious code to a benign file
  2. Creates an in-memory image from the file
  3. Creates a process from the image
  4. **Overwrites the file to make it benign**
  5. Creates the first thread
  6. Closes the file

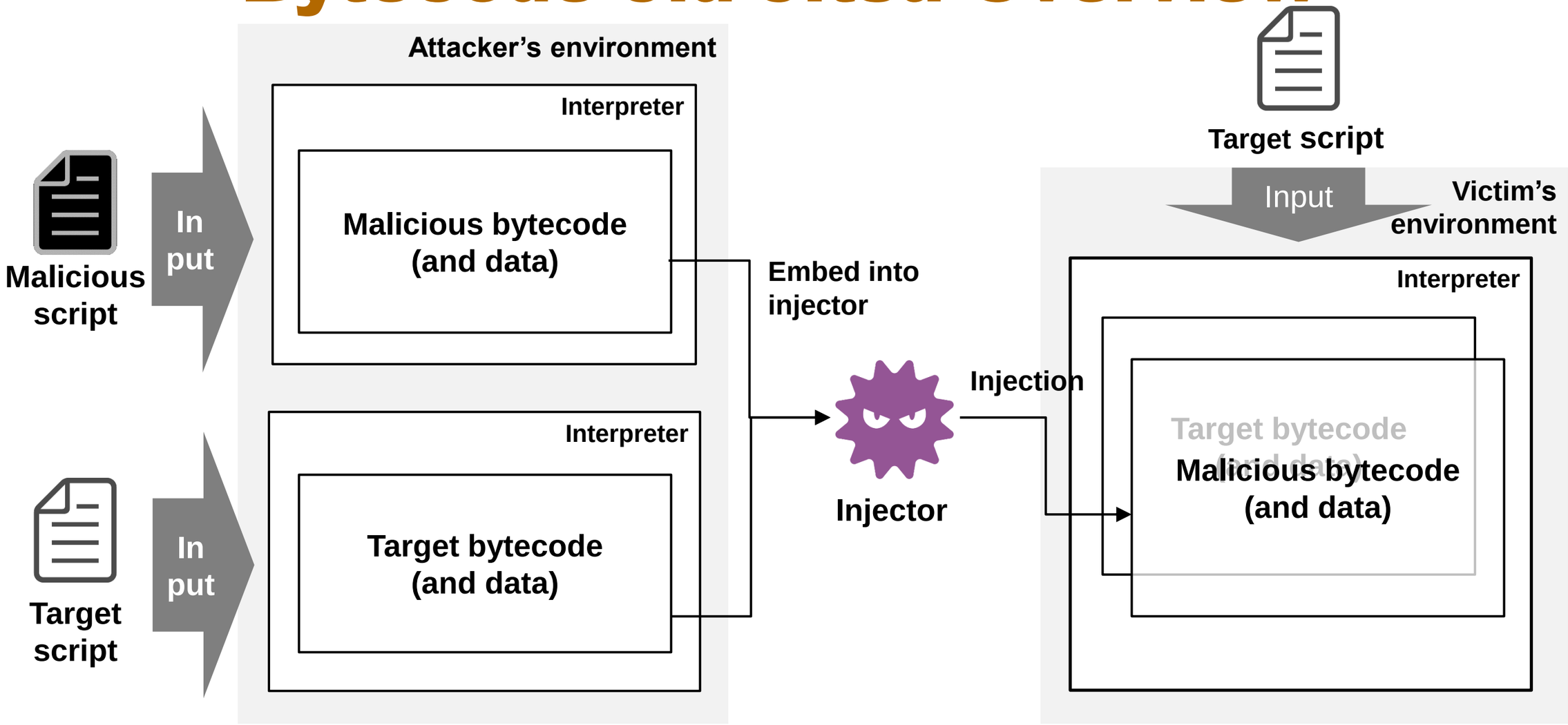
理令 Bytecode Jiu-Jitsu Overview

Our New Technique: Bytecode Jiu-Jitsu

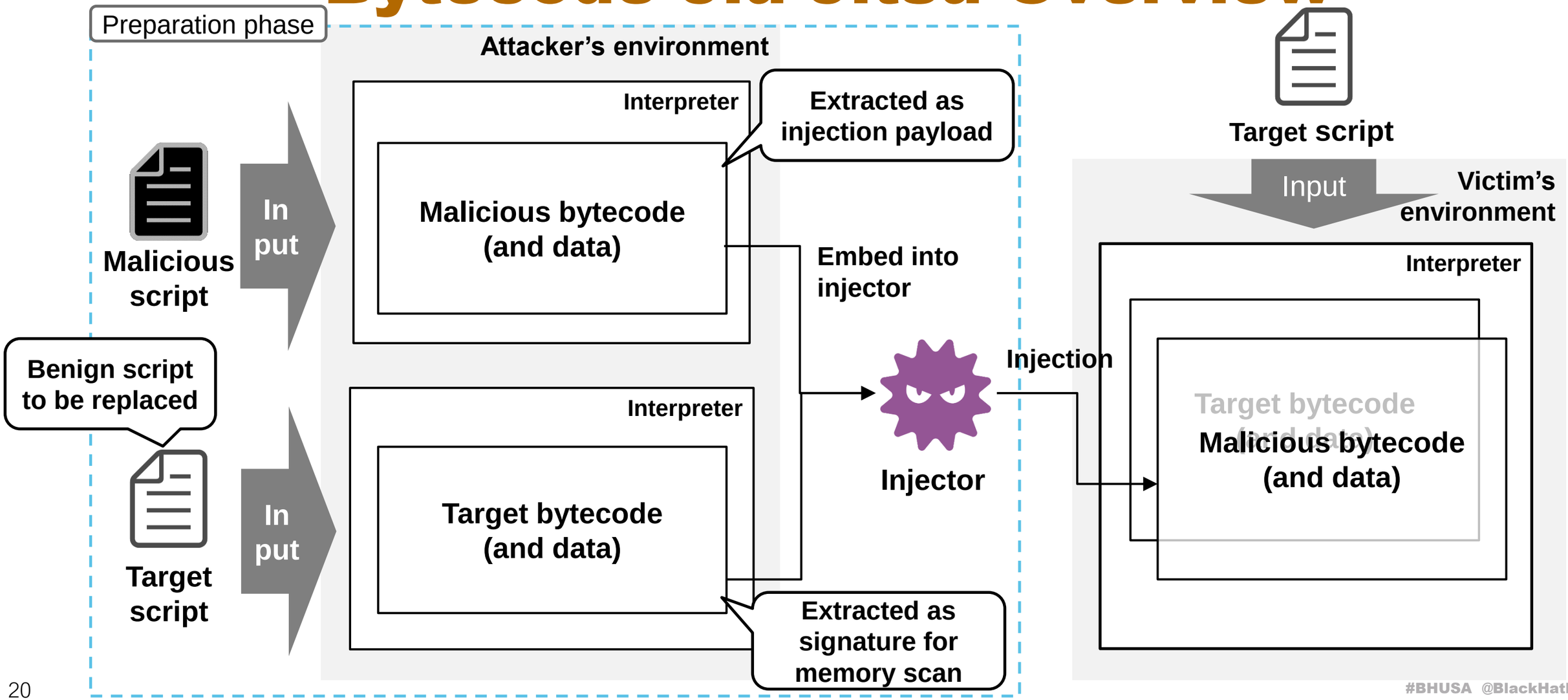
- We introduce a novel technique of a code injection attack
  ⇒ We call it ***Bytecode Jiu-Jitsu***
- The attack technique injects malicious ***bytecode*** into an interpreter process (e.g. Python)

	Existing attack techniques	Bytecode Jiu-Jitsu
Injection target	Arbitrary process	Interpreter process
Code to be injected	Native code	Bytecode
Behavior blended into	Executable	Script

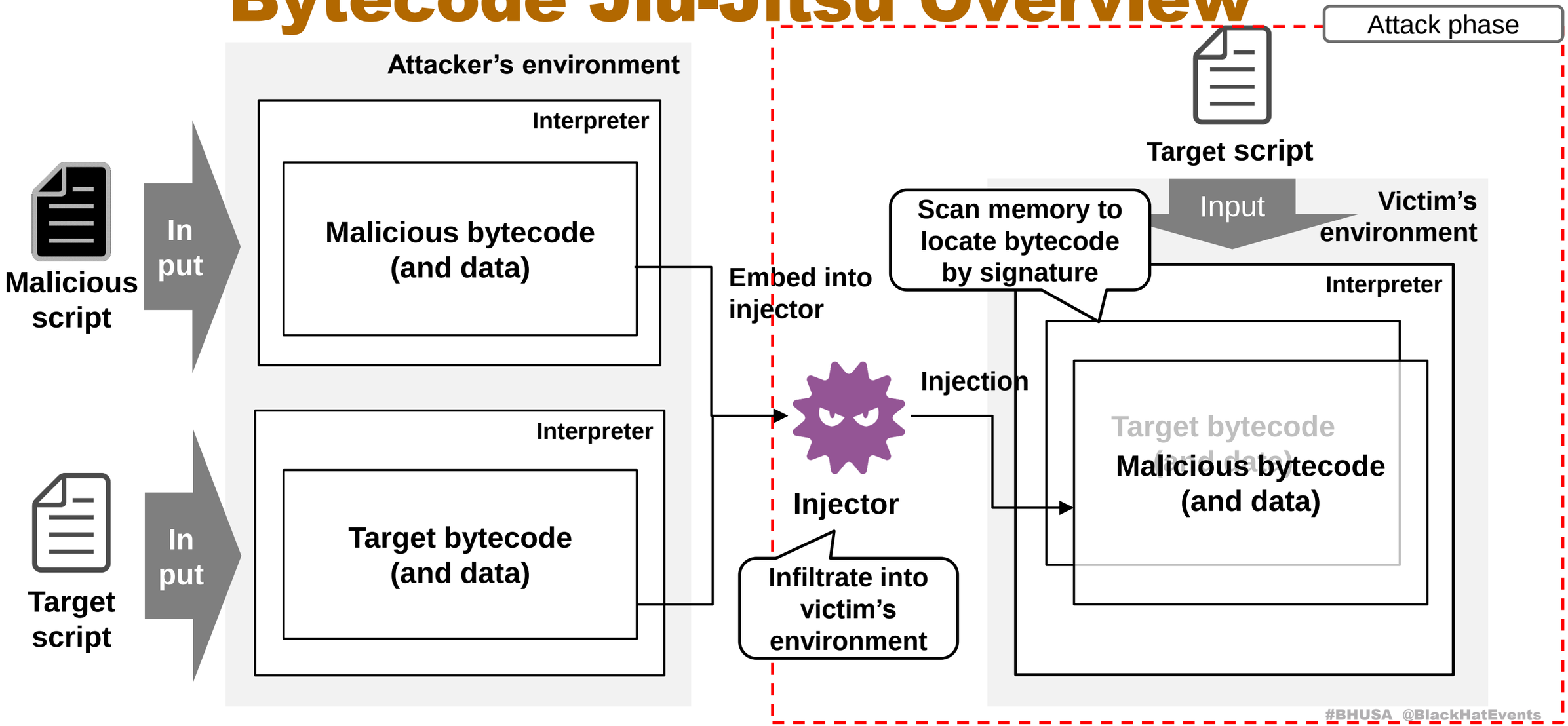
Bytecode Jiu-Jitsu Overview



Bytecode Jiu-Jitsu Overview



Bytecode Jiu-Jitsu Overview



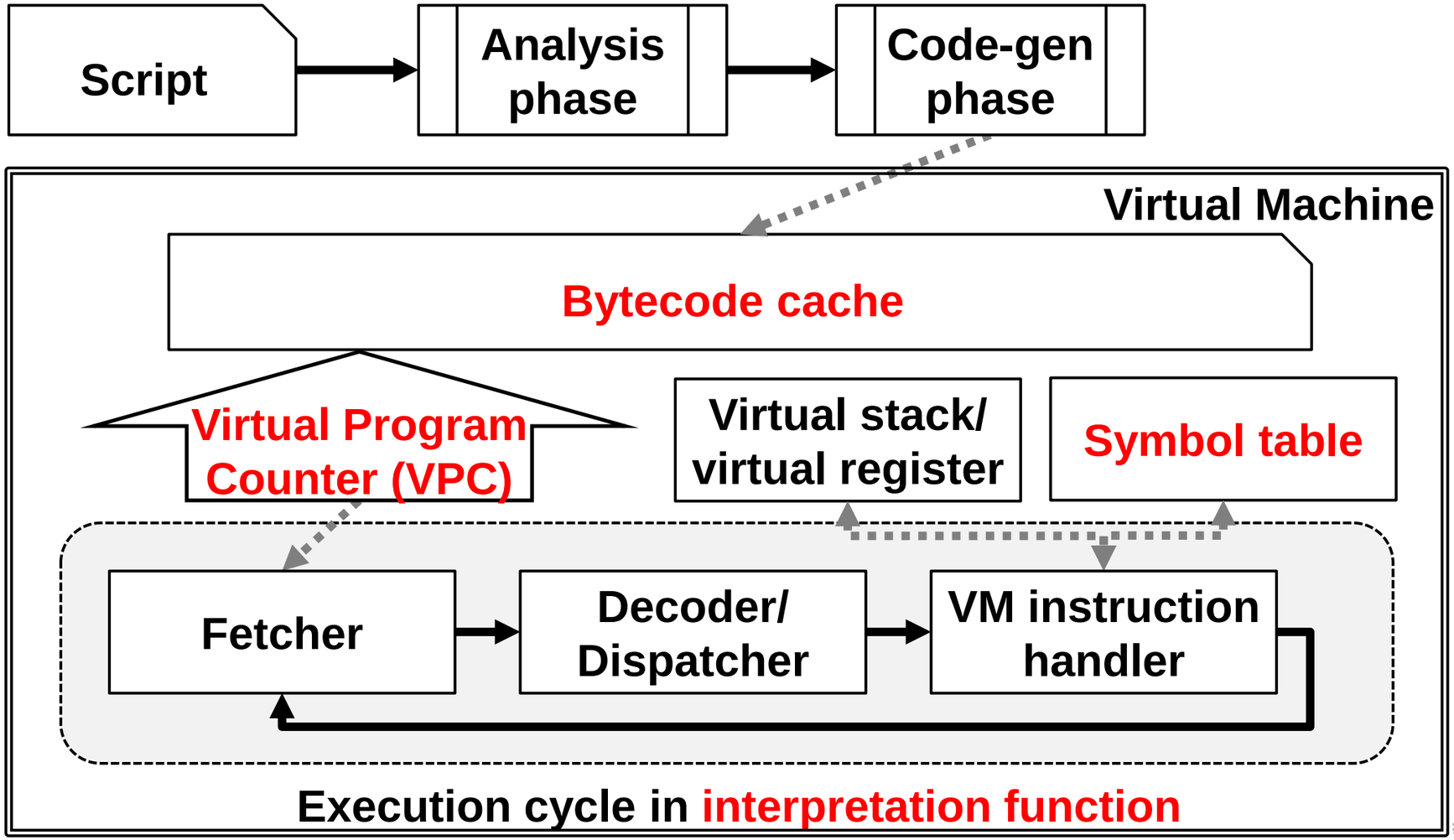
How to realize Bytecode Jiu-Jitsu?

- **Problem**
  - Bytecode Jiu-Jitsu requires the internal specifications of target interpreters
    i.e., **data structures of bytecode and data**
  - However, they are sometimes not publicly available

- **Solution:** Manual reverse engineering…?? 🤔

稽古 Interpreter Implementation Basics

Script Execution Mechanism



Bytecode Cache Implementation

Typically implemented with **array of structures**

Bytecode

```
…
LOAD_CONST      1
STORE_FAST      0
LOAD_FAST       0
LOAD_CONST      2
COMPARE_OP      2
POP_TOP
LOAD_CONST      0
…
```

Array of structures {Opcode, Operand}

Opcode	Operand
…	
LOAD_CONST	1
STORE_FAST	0
LOAD_FAST	0
COMPARE_OP	2
POP_TOP	2
LOAD_CONST	0
…	

Bytecode Cache Implementation

Typically implemented with **array of structures**

Bytecode

```
…
LOAD_CONST      1
STORE_FAST      0
LOAD_FAST       0
LOAD_CONST      2
COMPARE_OP      2
POP_TOP
LOAD_CONST      0
…
```

Array of st

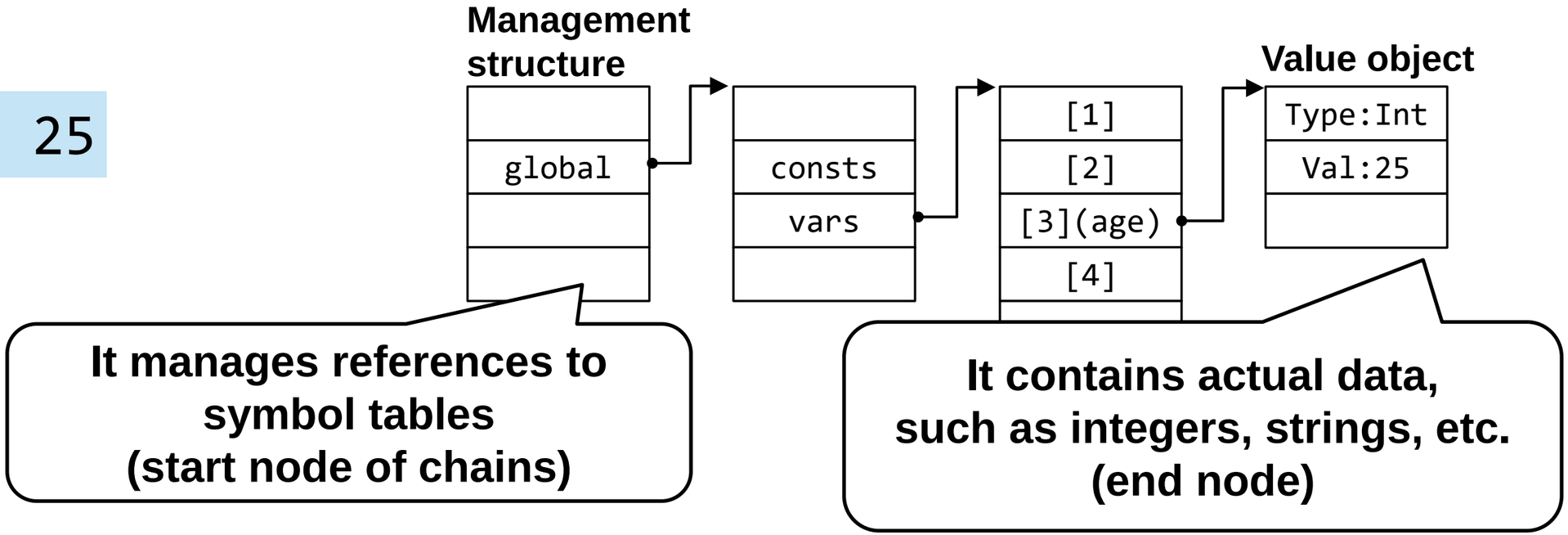
Opcod	
…	
LOAD_CONST	**1**
STORE_FAST	0
LOAD_FAST	0
COMPARE_OP	2
POP_TOP	2
LOAD_CONST	0
…	

Index for a symbol table
(Bytecode depends on symbol tables for data access.)

Symbol Table Implementation

Symbol tables are composed of references between multiple structures and arrays

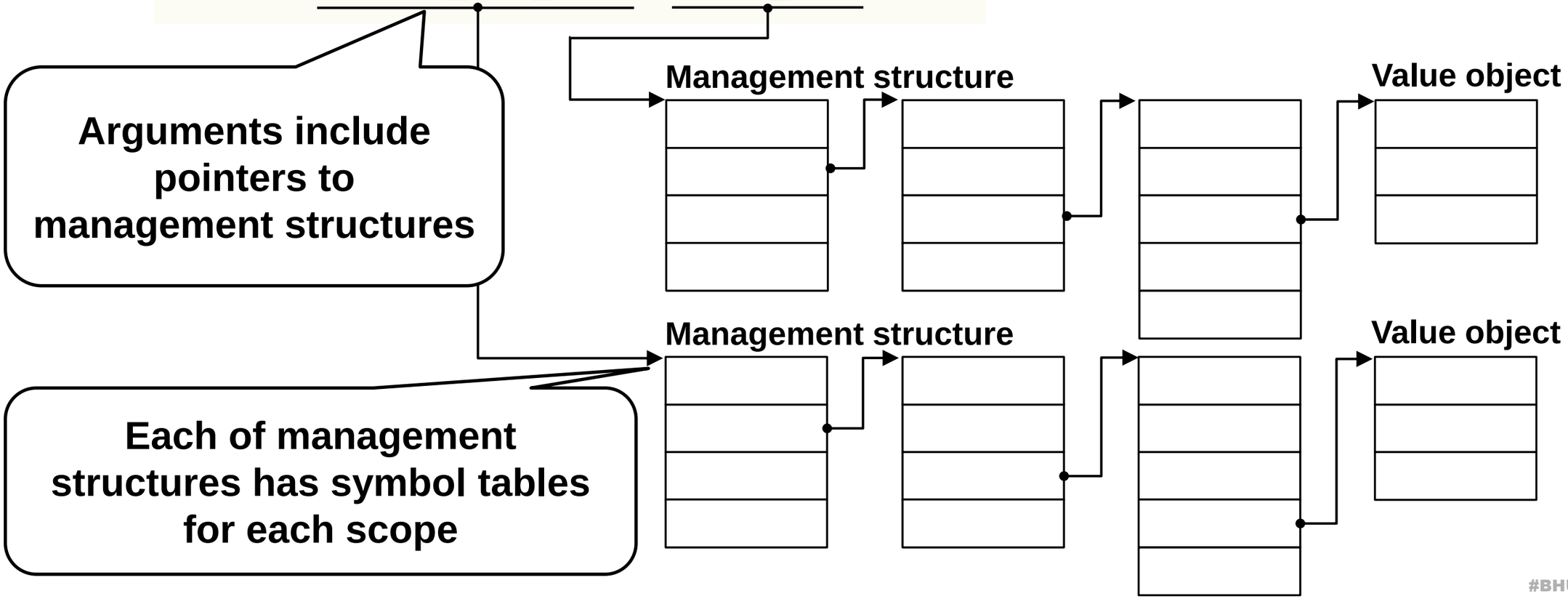
```
age = 25
```



Symbol Table Implementation

Interpretation function

```
interp(script_ctx_info, func_info, …)
```



Interpreter Analysis Issues

- These data structures are complicated.
  - Not easy to extract them because bytecode and symbol tables must be kept consistency between them.
- Interpreters share this overall design, but the concrete implementation details differ across interpreters and versions.

- Manual reverse engineering of interpreters requires heavy effort.
- Which means Bytecode Jiu-Jitsu is not practical …?

Interpreter Analysis Issues

- These data structures are complicated.
  - Not easy to extract them because bytecode and symbol tables must be kept consistency between them.
- Interpreters share this overall design, but the concrete implementation details differ across interpreters and versions.

- Manual reverse engineering of interpreters requires heavy effort.
- Which means Bytecode Jiu-Jitsu is not practical …?

  → **No, the reverse engineering can be automated!**

How to realize Bytecode Jiu-Jitsu?

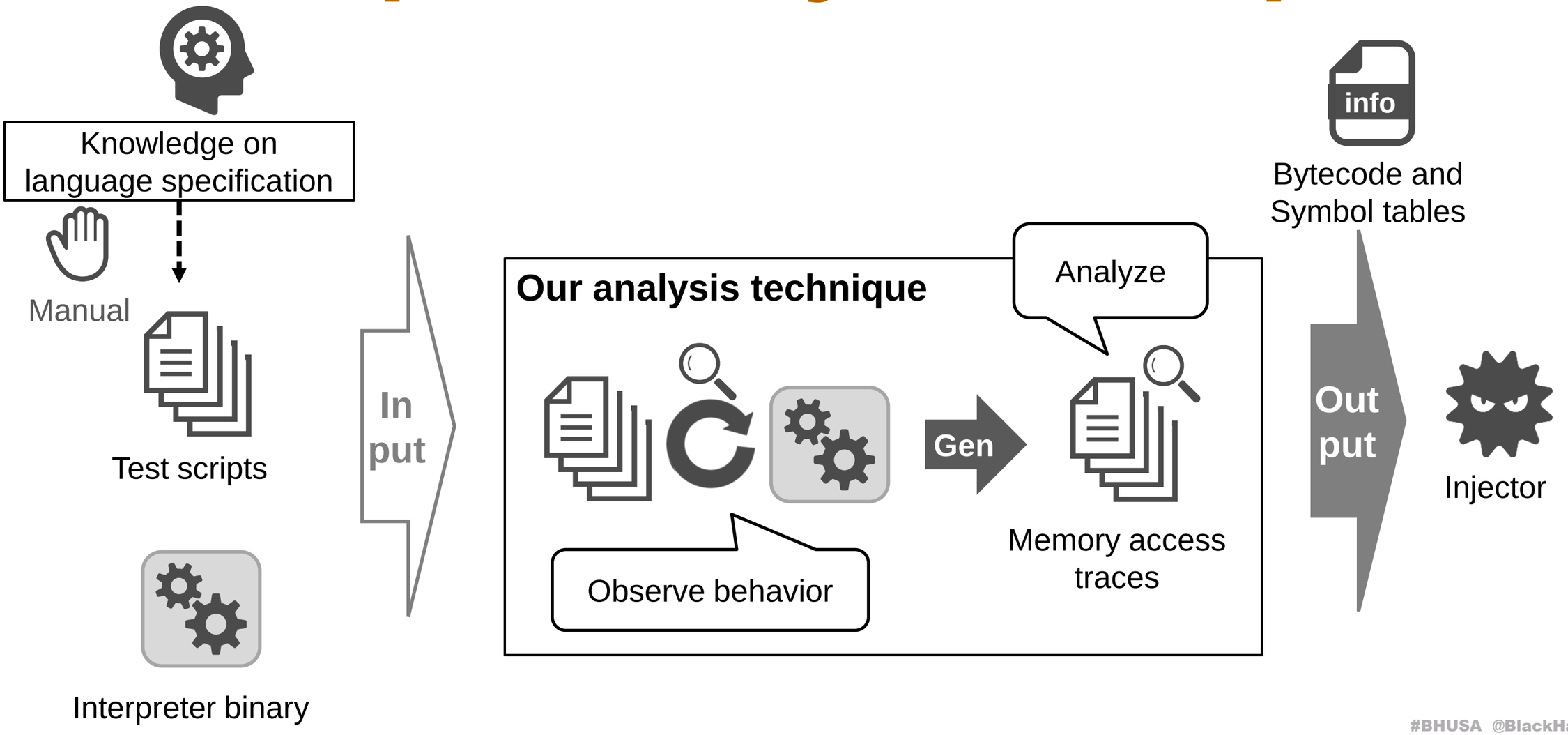
- **Problem**
  - Bytecode Jiu-Jitsu requires the internal specifications of target interpreters i.e., **data structures of bytecode and data**
  - However, they are sometimes not publicly available

Too tedious 😱

- **Solution:** ~~Manual reverse engineering…??~~
  → **Automated reverse engineering!!**
  - **Dynamic analysis of interpreter binaries by crafted testing scripts** for analyzing implementation details
  - **Tracking pointer dereferences and analyzing memory accesses** to reveal reference relationships and data structures

打边 Interpreter Analysis:
Prepare Bytecode and Symbol Tables to Inject

Interpreter Analysis Technique



Technical Overview

Interpretation function

```
interp(script_ctx_info, …)
```

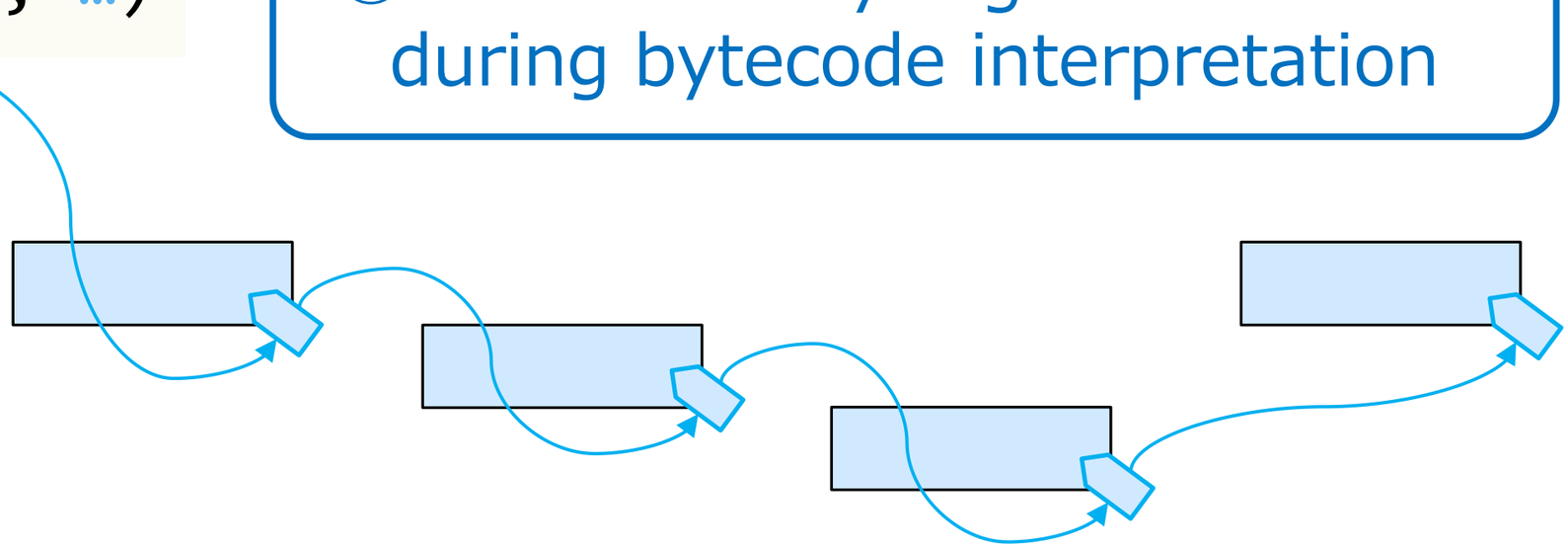
① Find the interpretation function

Technical Overview

Interpretation function

```
interp(script_ctx_info, …)
```

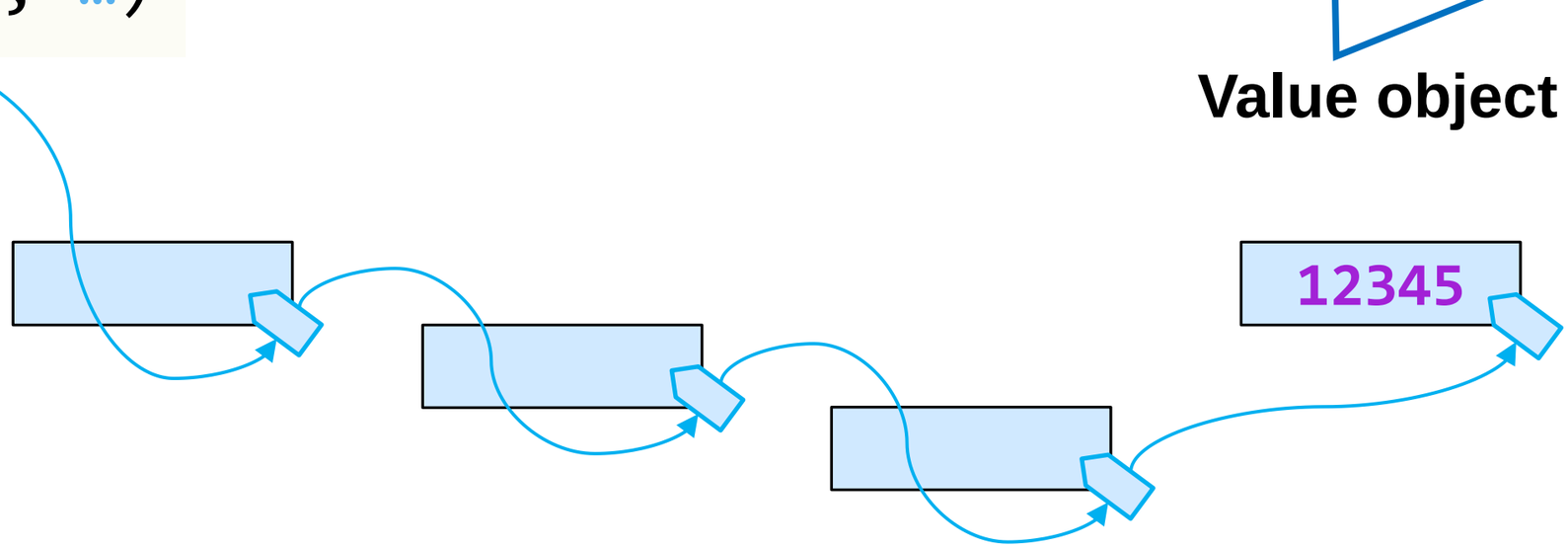
② Find memory regions accessed during bytecode interpretation



Technical Overview

Interpretation function

```
interp(script_ctx_info, …)
```

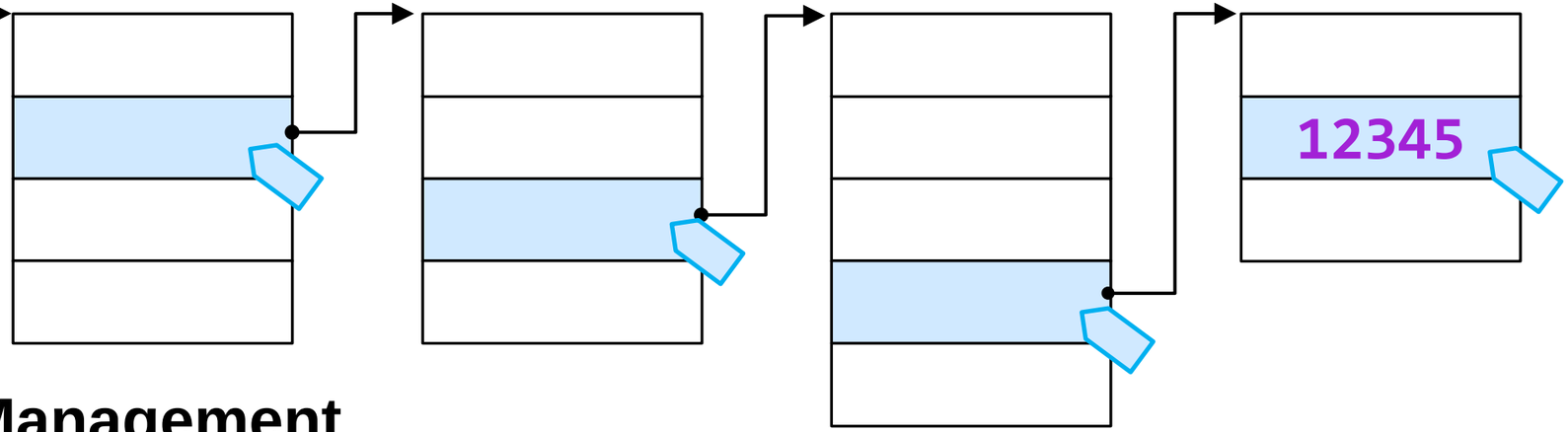


Technical Overview

Interpretation function

```
interp(script_ctx_info, …)
```

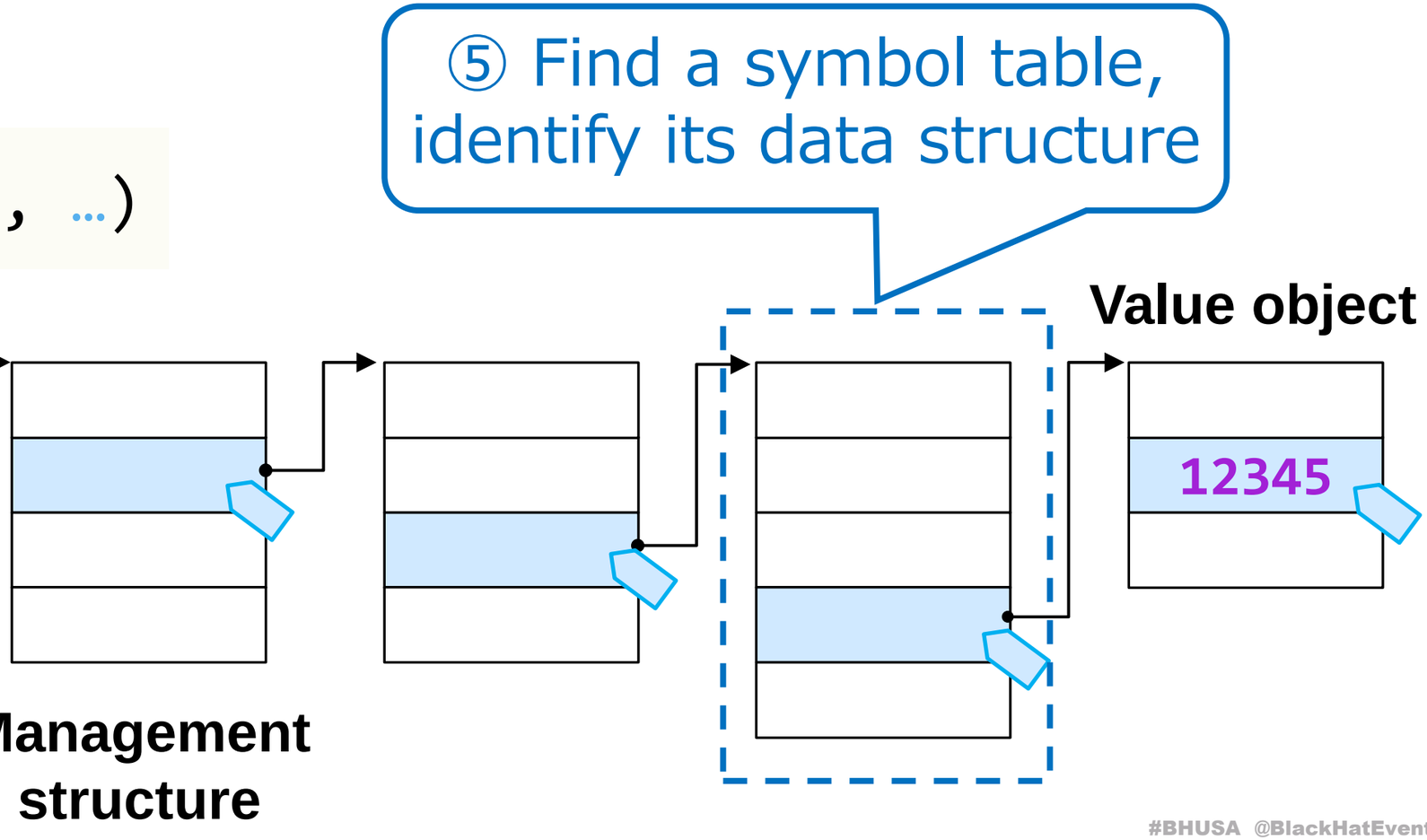
④ Find a dereference path to the object



Technical Overview

Interpretation function

```
interp(script_ctx_info, …)
```



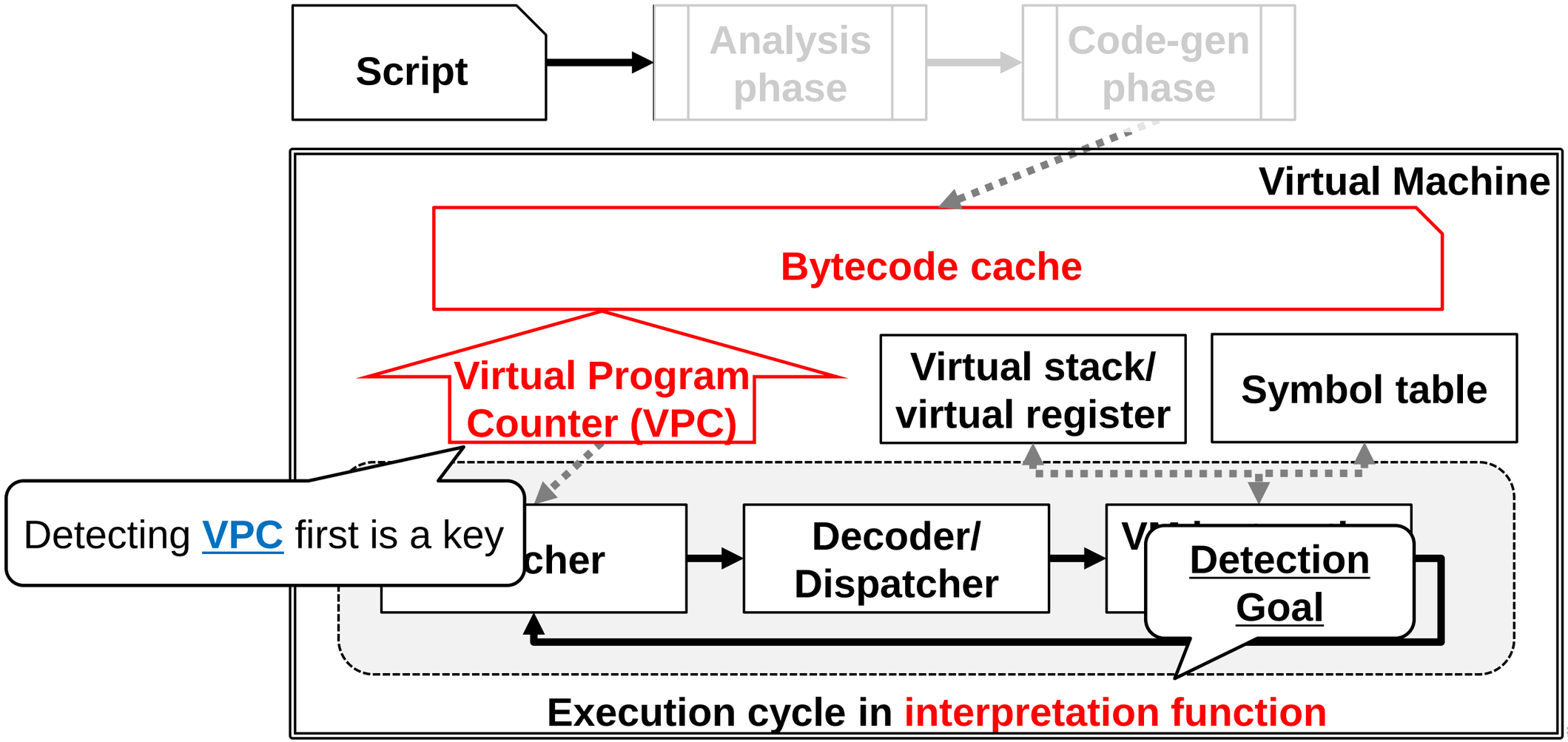
Key Steps of Interpreter Analysis

- Find the interpretation function
- Find accessed memory regions
- Find a value object
- Find a dereference path to the object
- Find a symbol table, identify its data structure
- Extract bytecode and symbol tables

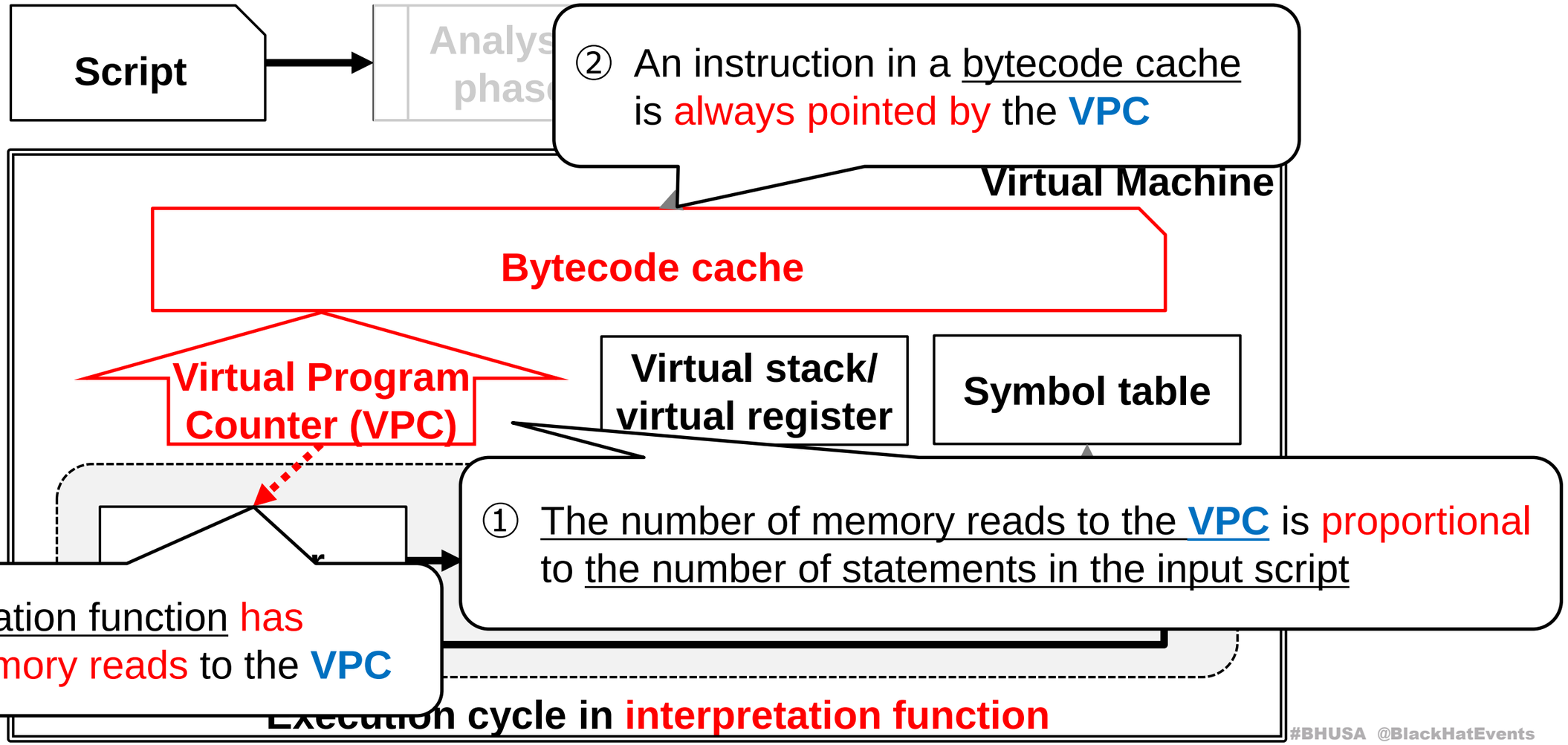
Key Steps of Interpreter Analysis

- Find the interpretation function
- Find accessed memory regions
- Find a value object
- Find a dereference path to the object
- Find a symbol table, identify its data structure
- Extract bytecode and symbol tables

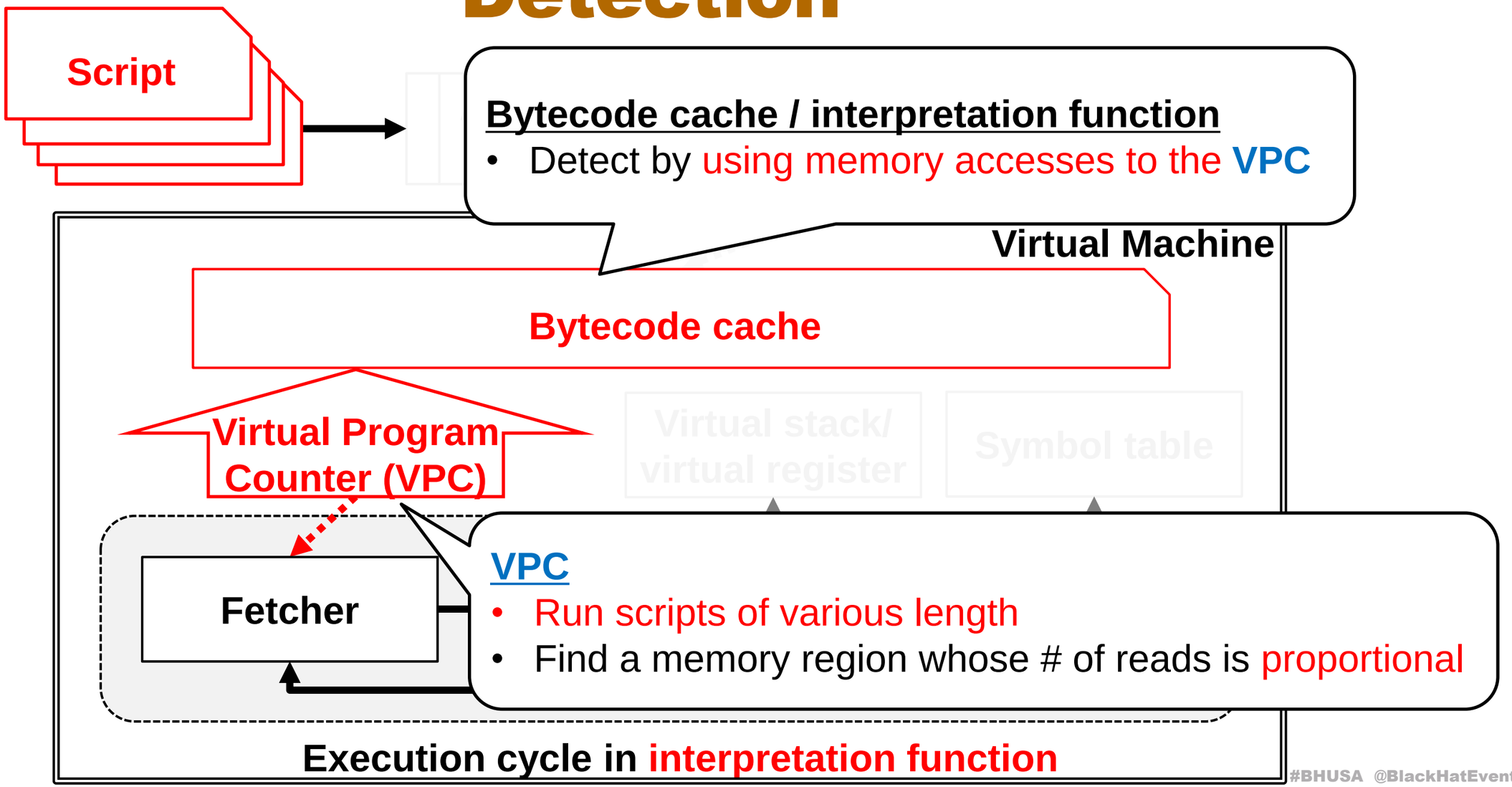
What do we need to know first?



Key Assumptions for Detection



Detection

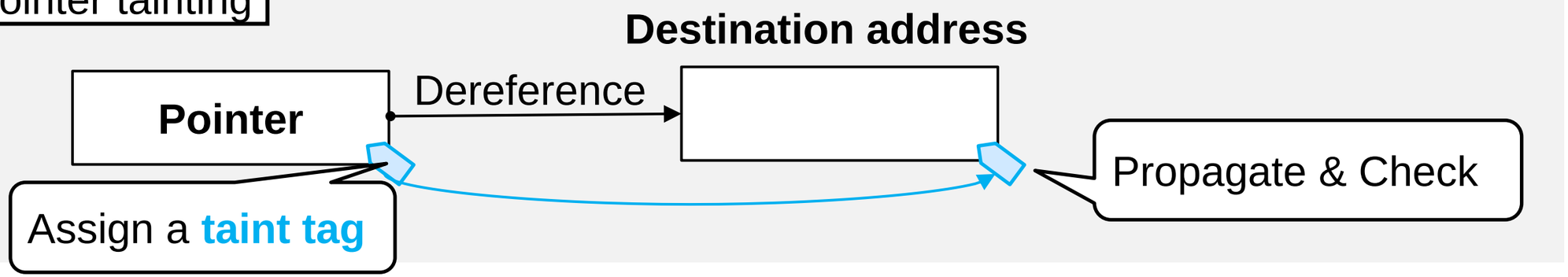

Script
Bytecode cache / interpretation function
• Detect by using memory accesses to the VPC
Virtual Machine
Bytecode cache
Virtual Program Counter (VPC)
Virtual stack/ virtual register
Symbol table
Fetcher
VPC
• Run scripts of various length
• Find a memory region whose # of reads is proportional
Execution cycle in interpretation function

Key Steps of Interpreter Analysis

- Find the interpretation function
- Find accessed memory regions
- Find a value object
- Find a dereference path to the object
- Find a symbol table, identify its data structure
- Extract bytecode and symbol tables

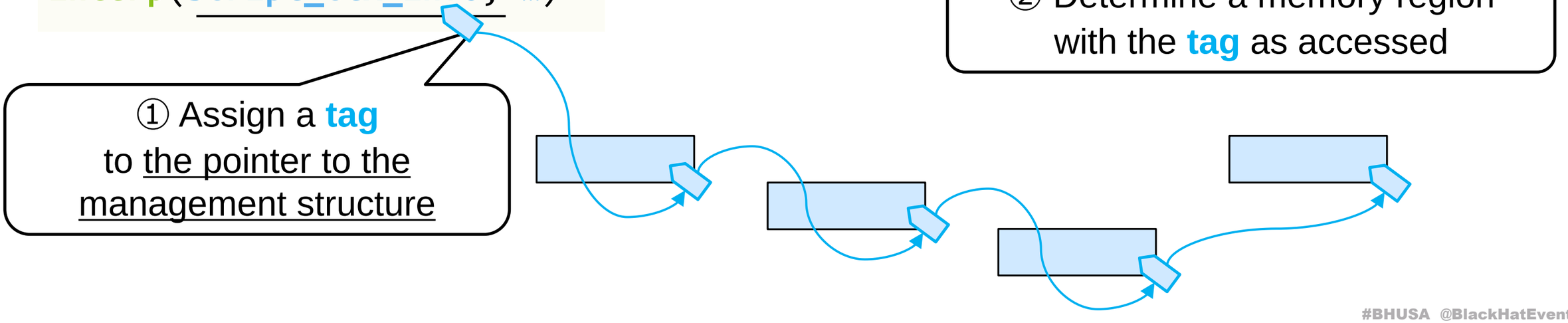
Accessed Memory Region Detection

Pointer tainting

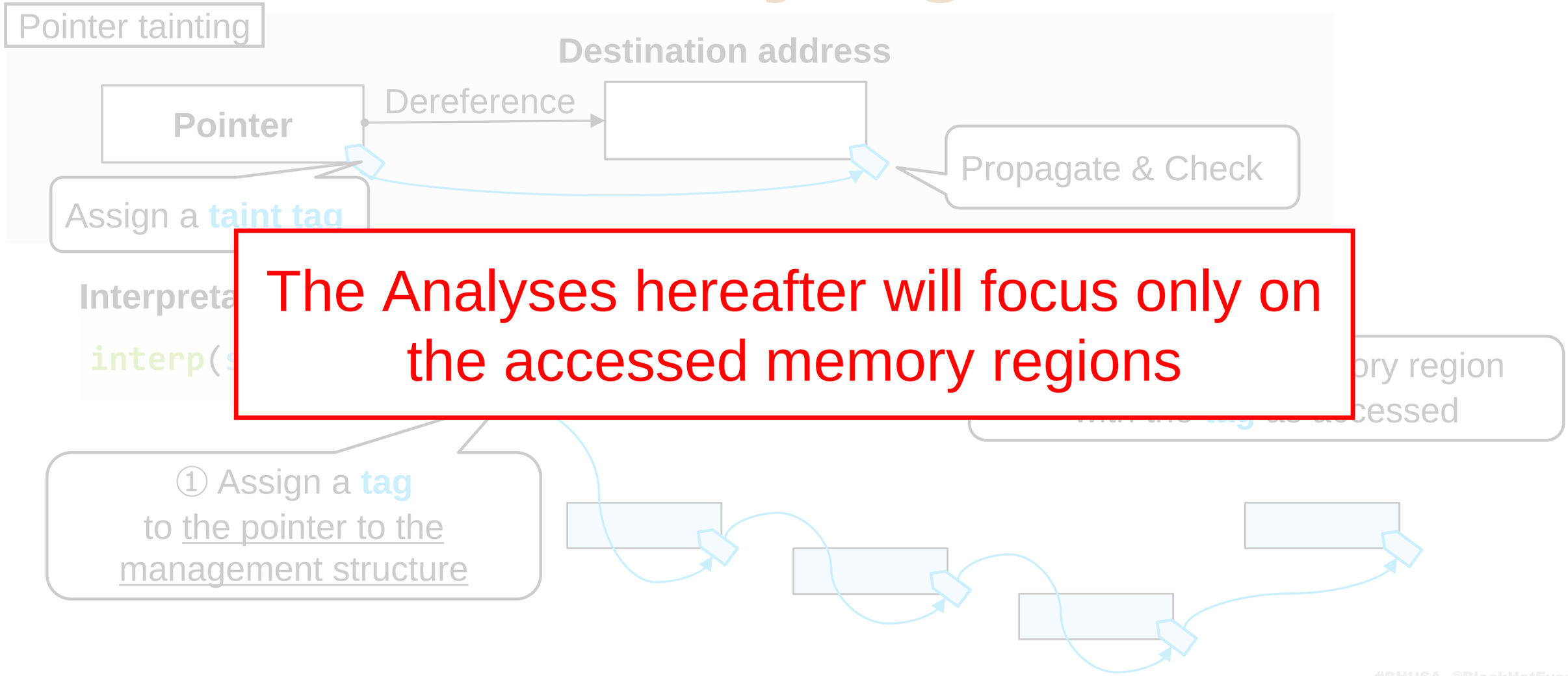


Interpretation function

```
interp(script_ctx_info, …)
```



Accessed Memory Region Detection



Key Steps of Interpreter Analysis

- Find the interpretation function
- Find accessed memory regions
- Find a value object
- Find a dereference path to the object
- Find a symbol table, identify its data structure
- Extract bytecode and symbol tables

Features of Test Script

- We manually craft test scripts to:
  - Run dynamic analysis
  - Control the memory state for the convenience of later analysis

```
global_var = 123456
```

Feature 2: Use a characteristic value searchable in memory

Feature 1: Has an assignment statement in each scope
(this example is for global scope)

Value Object Detection

Test script

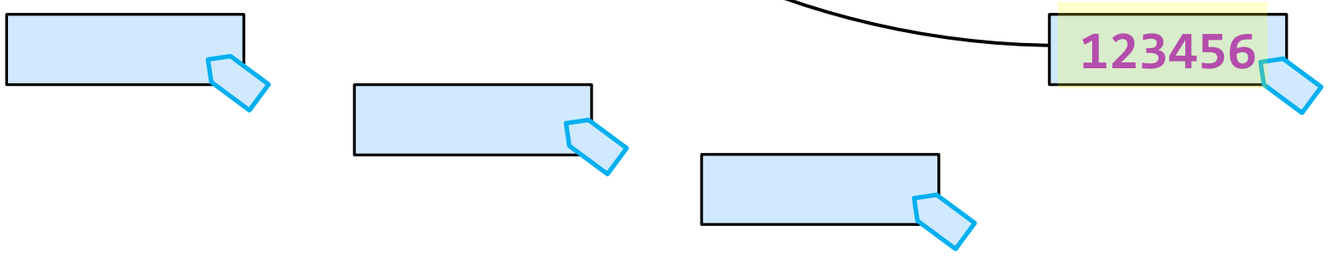
```
global_var = 123456
```

Find a value object by searching memory for a **characteristic value**

Interpretation function

```
interp(script_ctx_info, …)
```

Value object

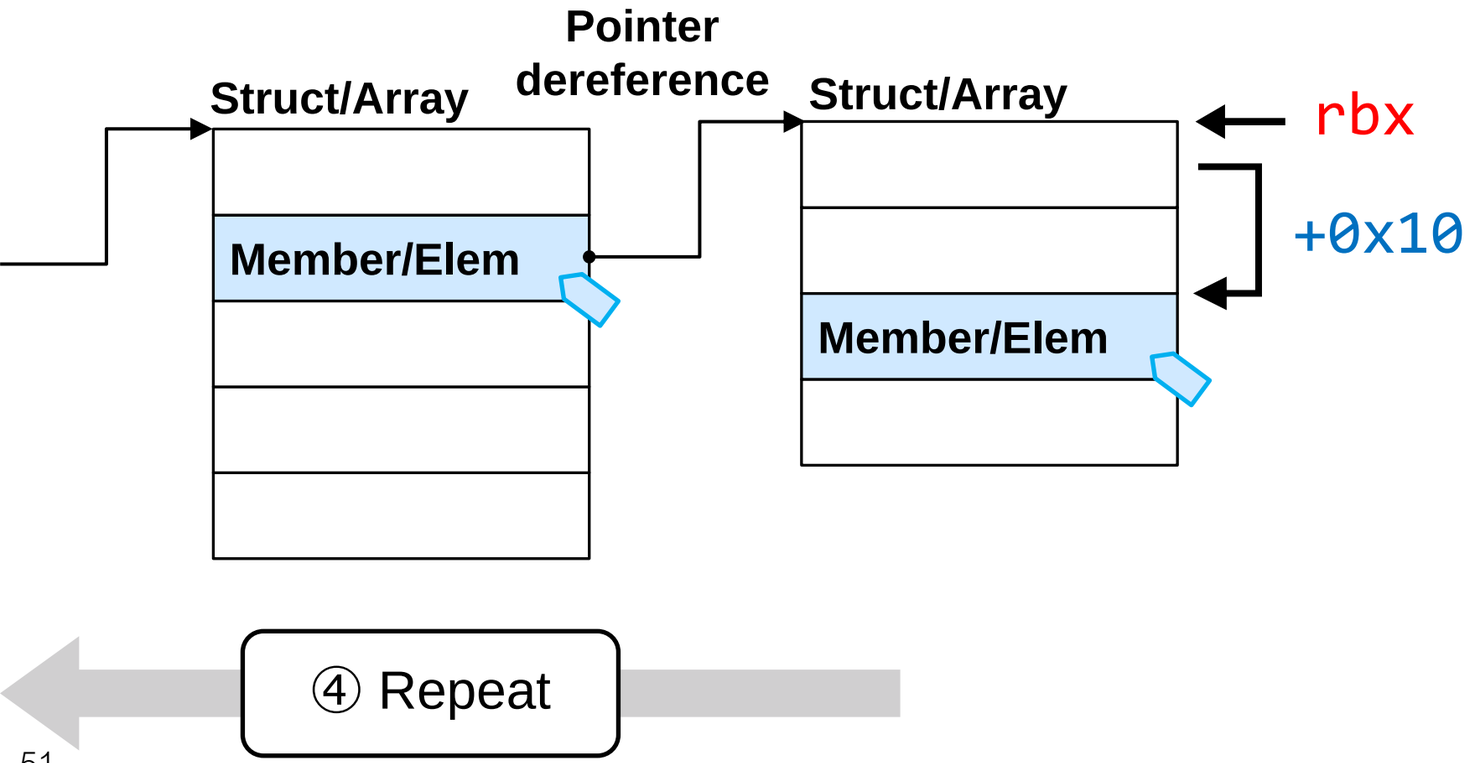


Key Steps of Interpreter Analysis

- Find the interpretation function
- Find accessed memory regions
- Find a value object
- Find a dereference path to the object
- Find a symbol table, identify its data structure
- Extract bytecode and symbol tables

Structure/Array Dereference Analysis

- Find structure/array accesses
- Determine base addresses and offsets




```
mov rcx, [ rax + 0x40  ]
mov rbx, [ rcx + rsi*8 ]
mov rax, [ rbx + 0x10  ]
```

① Find memory accesses that use the base register

② Get base address

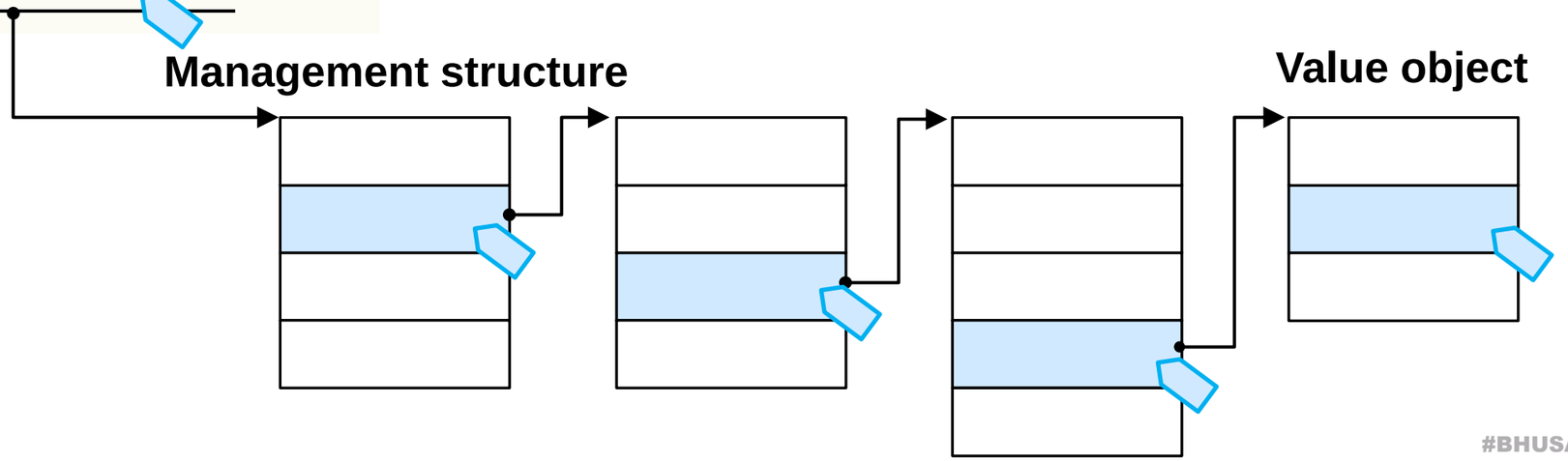
③ Get offset/index

Dereference Analysis of Symbol Tables

- Analyze all accessed structures and arrays
- Find **dereference paths from the management structure to value objects**

Interpretation function

```
interp(script_ctx_info, …)
```



Key Steps of Interpreter Analysis

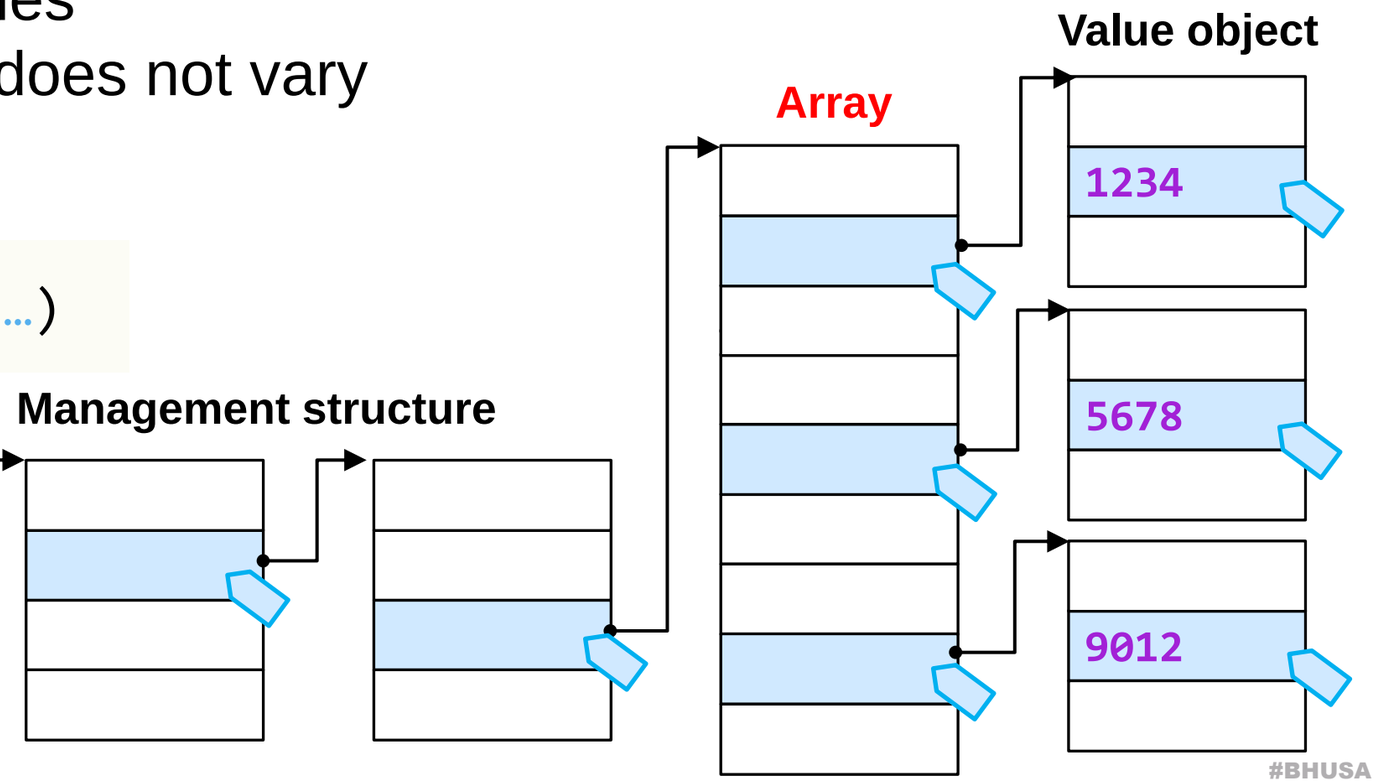
- Find the interpretation function
- Find accessed memory regions
- Find a value object
- Find a dereference path to the object
- Find a symbol table, identify its data structure
- Extract bytecode and symbol tables

Structure Analysis of Symbol Tables

- A symbol table containing arbitrary number of variables must be handled
- If references to value objects in the symbol table are managed with **arrays**
  ⇒ Array length only varies
  ⇒ Reference structure does not vary

Interpretation function

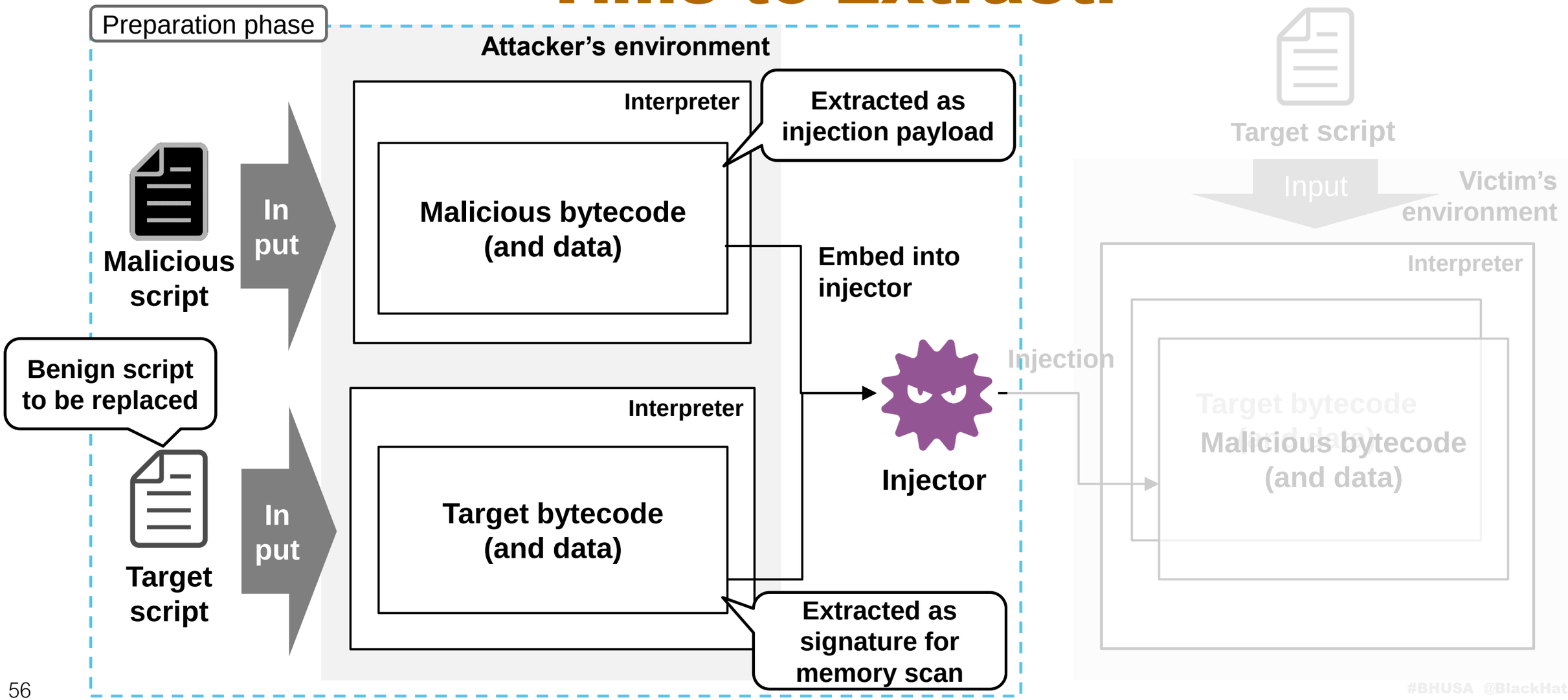
```
interp(script_ctx_info, …)
```



Key Steps of Interpreter Analysis

- Find the interpretation function
- Find accessed memory regions
- Find a value object
- Find a dereference path to the object
- Find a symbol table, identify its data structure
- Extract bytecode and symbol tables

Time to Extract!



Extraction of Bytecode and Symbol Tables

① Execute a malicious script with the behavior to inject

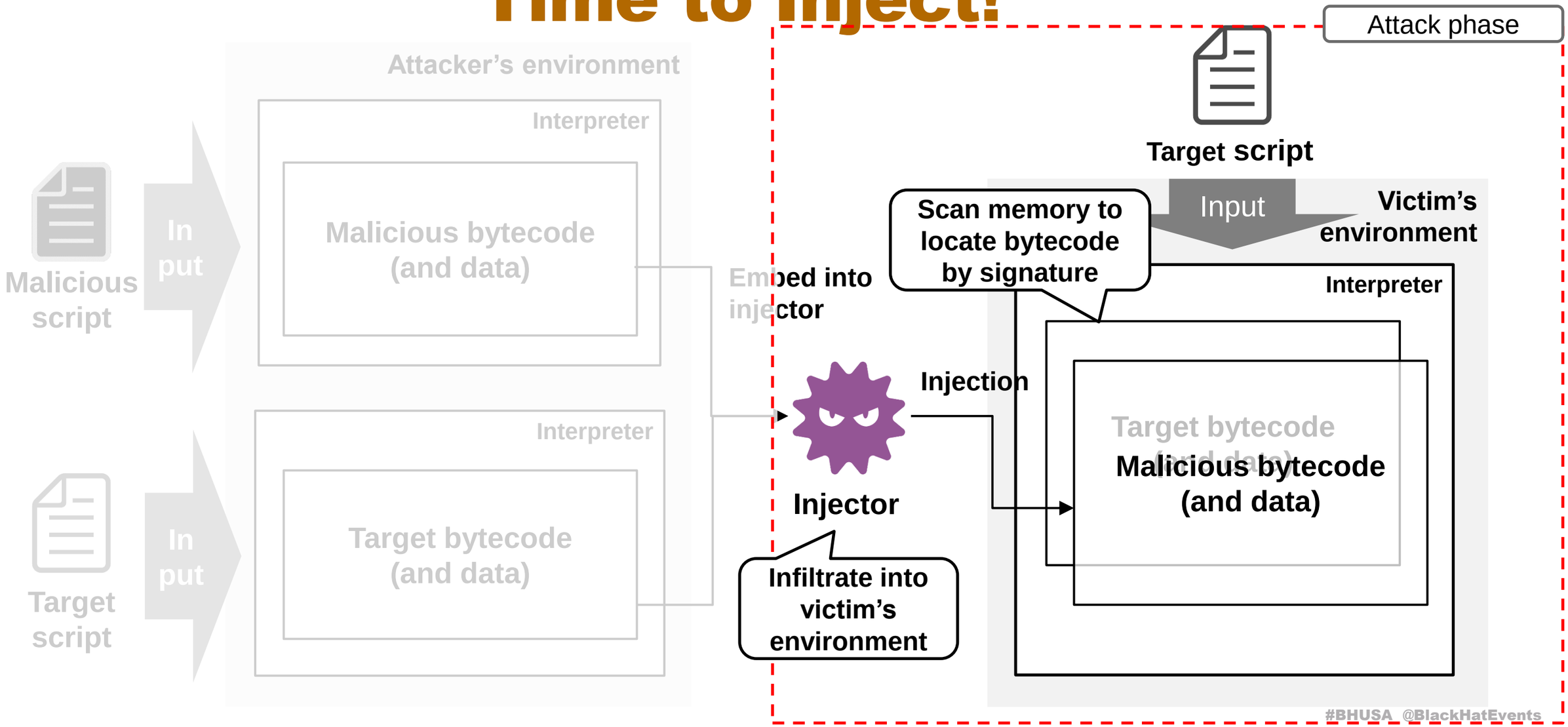
② Suspend the execution at the beginning of the interpretation function

③ Explore the structures from the management structure to symbol tables based on the obtained structural information

④ Read their memory to extract bytecode and symbol tables

試合 Bytecode Jiu-Jitsu Attack:
Determine Place to Inject in Victim's Environment

Time to Inject!



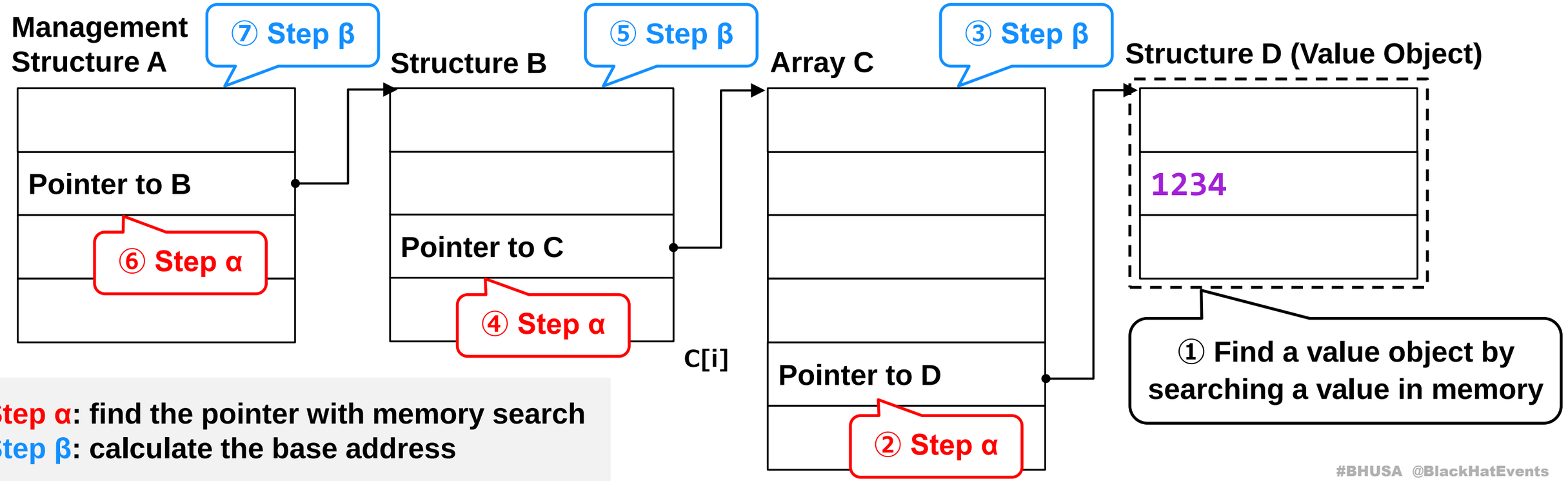
Know Your Victim

- **Final step: Locate the proper position to inject to**
  - Memory space layout is randomized
    - The location of bytecode and symbol tables differs across executions
  - It is difficult to reveal the internal memory state of the interpreter in the victim's environment
    - Should not use debuggers because it's too suspicious

- **Approach: memory search and exploration**
  - Identify internal state by memory read only
    - Without using debuggers

Recognizing Structure of Target Interpreter

1. Suspend execution and enumerate all stack and heap memory
2. Detect management structures by backtracking from a value object



Step α: find the pointer with memory search
Step β: calculate the base address

Injection of Bytecode and Symbol Tables

① Traverse memory in the forward direction

② Write bytecode and symbol tables

③ Overwrite the VPC to point to the bytecode entry

④ Resume the execution

乱取 Experiments and Evaluations

Experimental Setup

Chose open-source interpreters as targets to verify detection points

Target interpreters	Feature	Implementation type
Python	Widely used / Attackers frequently use	Open source
Lua		
VBScript		Both open source and proprietary

Analysis/Injection Test

Interpreters	VPC	Bytecode cache	Interp. function	Symbol tables		Value object
				Detection	Analysis	
Python	✓	✓	✓	✓	✓	✓
Lua	✓	✓	✓	✓	✓	✓
VBScript	✓	✓	✓	✓	✓	✓

Interpreters	Bytecode, symbol tables		Code execution
	Extraction	Injection	
Python	✓	✓	✓
Lua	✓	✓	✓
VBScript	✓	✓	✓

All steps of our analysis technique could analyze interpreters correctly

Detectability of Bytecode Jiu-Jitsu

- We built two types of Bytecode Jiu-Jitsu injectors
  - Inject **infinite loop**: for evaluating detectability of just the injection behavior
  - Inject **downloader malware**: for evaluating detectability of injection + bytecode behavior

- Evaluated whether each security tool can detect them

Security tools	Tools used for the experiment
Anti-virus (AV)	72 AV products
Malware analysis sandbox	CAPE sandbox
Endpoint Detection and Response (EDR)	System monitoring tool (frequently used as simple EDR)
Memory forensics tools	Volatility with hollowfind/imgmalfind/ptemalfind

Detectability of Bytecode Jiu-Jitsu: Result

Security tools	Detection result	
	Infinite loop	Downloader
AV	9/72	9/72
Sandbox	✗	✓
EDR	✗	✓
Memory forensics tools	✗	✗

Detectability of Bytecode Jiu-Jitsu: Result

Security tools	Detection result	
	Infinite loop	Downloader
AV	9/72	9/72
Sandbox	✗	✓
		✓
Memory forensics tools	✗	✗

Only 9 AI-based engines flagged it as suspicious

Detectability of Bytecode Jiu-Jitsu: Result

Security tools	Detection result	
	Infinite loop	Downloader
AV	9/72	9/72
Sandbox	✗	✓
EDR	✗	✓
Memory forensics	✗	✗

- Injection requires only memory read/write, which makes it difficult to detect
- Detected the behavior of injected bytecode

Detectability of Bytecode Jiu-Jitsu: Result

Security tools	Detection result	
	Infinite loop	Downloader
AV	9/72	9/72
[redacted]	✗	✓
EDR	✗	✓
Memory forensics tools	✗	✗

- Detection relies executable permission of memory
- Bytecode Jiu-Jitsu does not require it and out of their scope

Demo


受身 Countermeasures against
Bytecode Jiu-Jitsu

Countermeasures with Existing Tools

- **AV**
  - Flag memory read/write APIs as suspicious
- **EDR and sandbox**
  - Detect memory writes to an interpreter process
  - Determine whether the written data is bytecode using signatures, etc.
- **Memory forensics**
  - Analyze an injector binary, detect unnatural parent-child relationships
- **OS security**
  - Protect interpreter processes and restrict memory write accesses
- **Manual analysis**
  - Difficult. No bytecode specification, debuggers, or disassemblers

Countermeasures in Future Studies

- **Bytecode / Malicious bytecode identification**

Identification	Input	Output	Applies to
Bytecode	Unknown byte sequence	Bytecode / Not	EDRs and sandboxes
Malicious bytecode	Bytecode	Malicious / Benign	Memory forensics

  - Learning-based approach may be applicable

- **Manual analysis support**
  - Analyze instruction set of bytecode, build debuggers/disassemblers

総括 Takeaways

Takeaways

- **Utilizing bytecode for code injection** had not been much discussed before
- **Our reverse engineering techniques** revealed it to be a **realistic threat**
  → **Be more careful about bytecode as payload** from now on!
- **Security researchers should discuss further countermeasures**
  – We wish our PoC tools will help them

Our PoC tools will be available soon here:
https://github.com/ntt-zerolab/Bytecode_Jiu-Jitsu

Thank you!