Detile: Fine-Grained Information-Leak Detection in Script Engines

Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, Thorsten Holz

Ruhr-Universität Bochum
Horst Görtz Institute for IT-Security
Bochum, Germany

Introduction

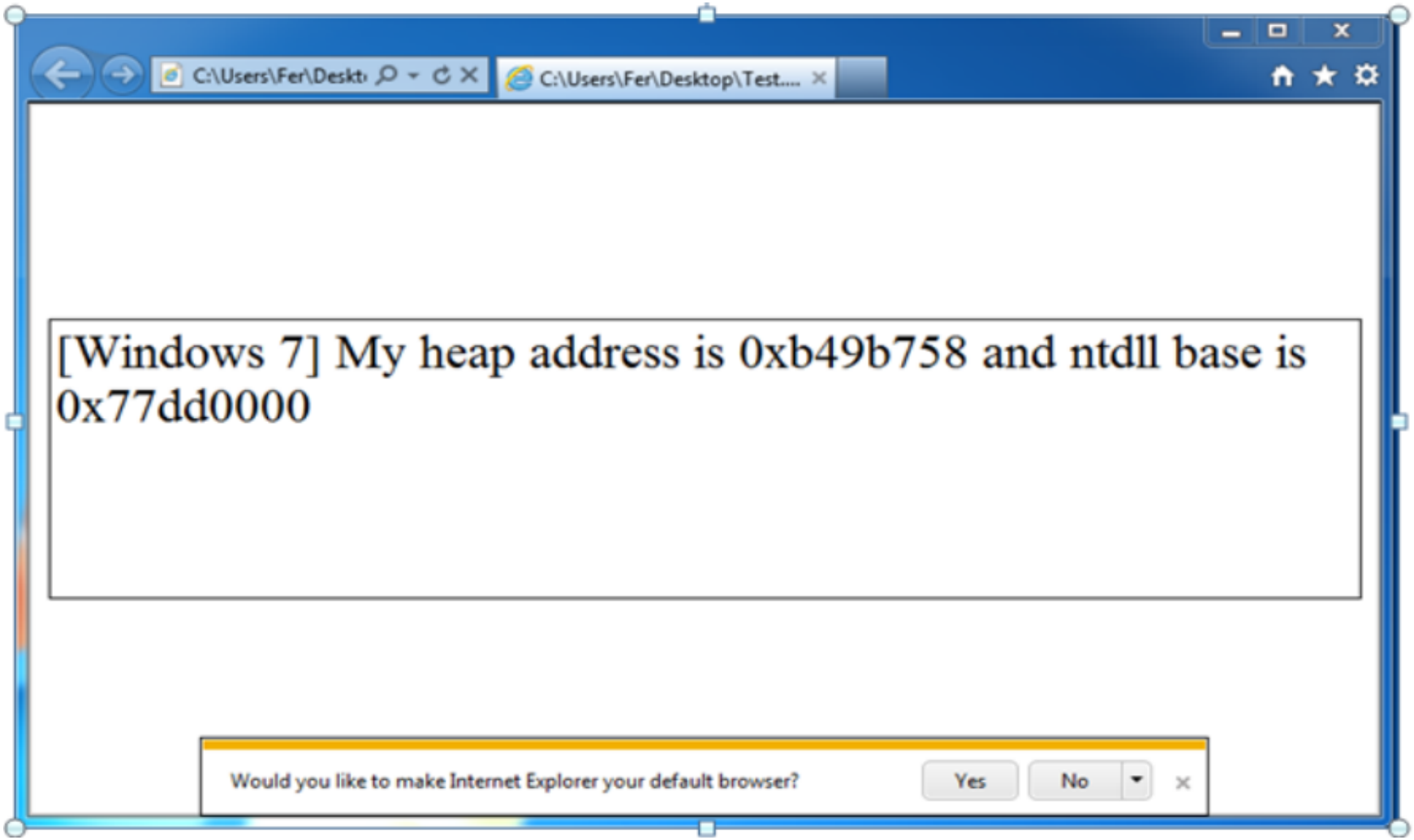
Introduction

- Web browsers are prone to memory corruption vulnerabilities
- Widely deployed exploit mitigations: DEP and ASLR
- Attackers ultimate goal: Execute code of choice via control-flow hijacking

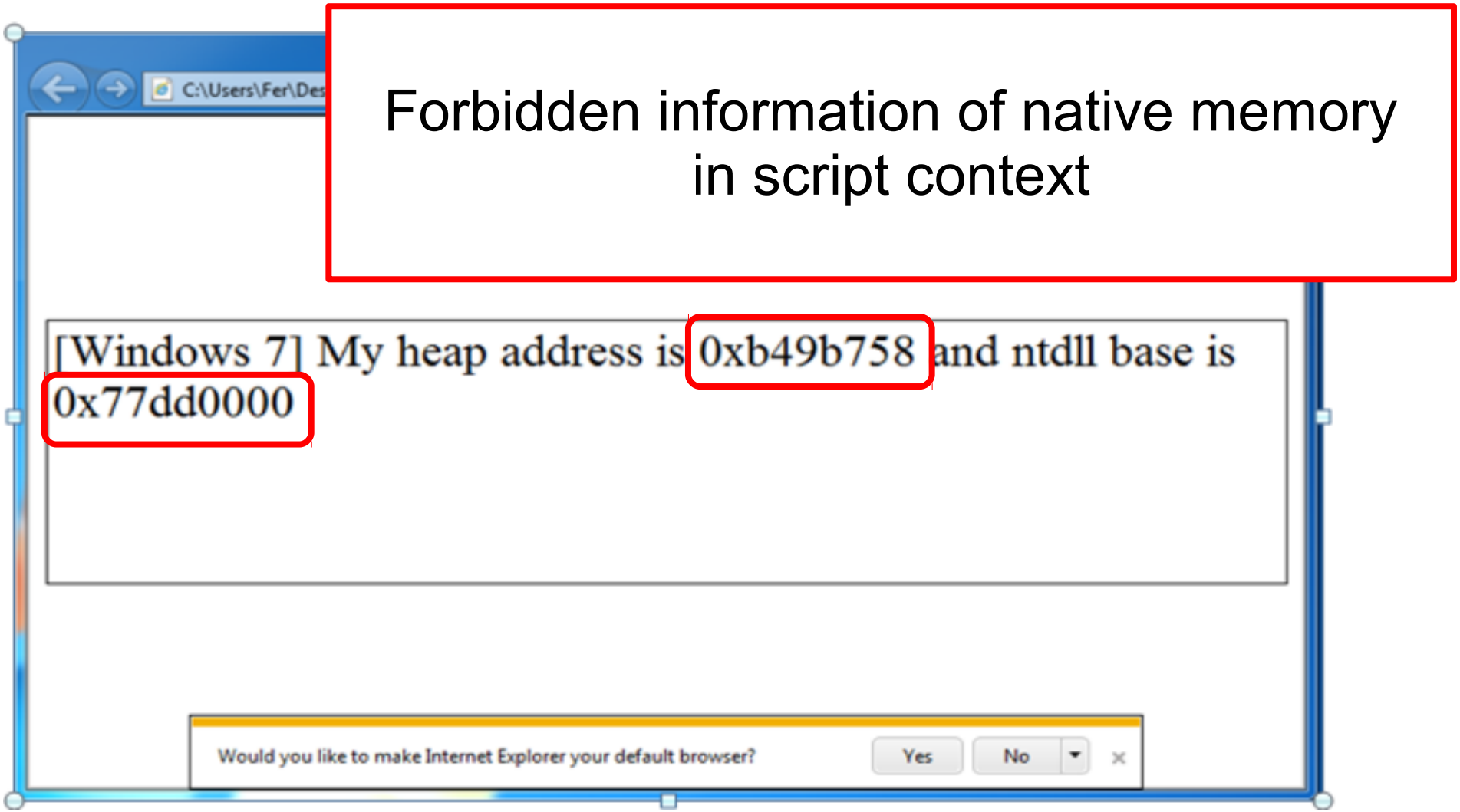
→ Knowledge about memory layout is necessary to bypass ASLR (*information leak* or *memory disclosure*)

Afterwards, code reuse can be conducted

Information leaks in-the-wild, e.g., CVE-2012-0769 [1]:



Information leaks in-the-wild, e.g., CVE-2012-0769 [1]:



Information leaks in academia:

- Defense: Fine-Grained ASLR [2]
→ Bypass: Just-In-Time Code Reuse [3]

- Defense: Destructive Code Reads [4]
→ Bypass: Code-Inference Attacks [5]

- Defense: G-Free [6]
→ Bypass: Browser JIT attacks [7]

… and many more

Background

Memory disclosures in browsers

- Uninitialized variables
  - JPEG parsing leaks stack addresses (CVE-2014-6355)
  - TIFF processing information leak (CVE-2015-0061)

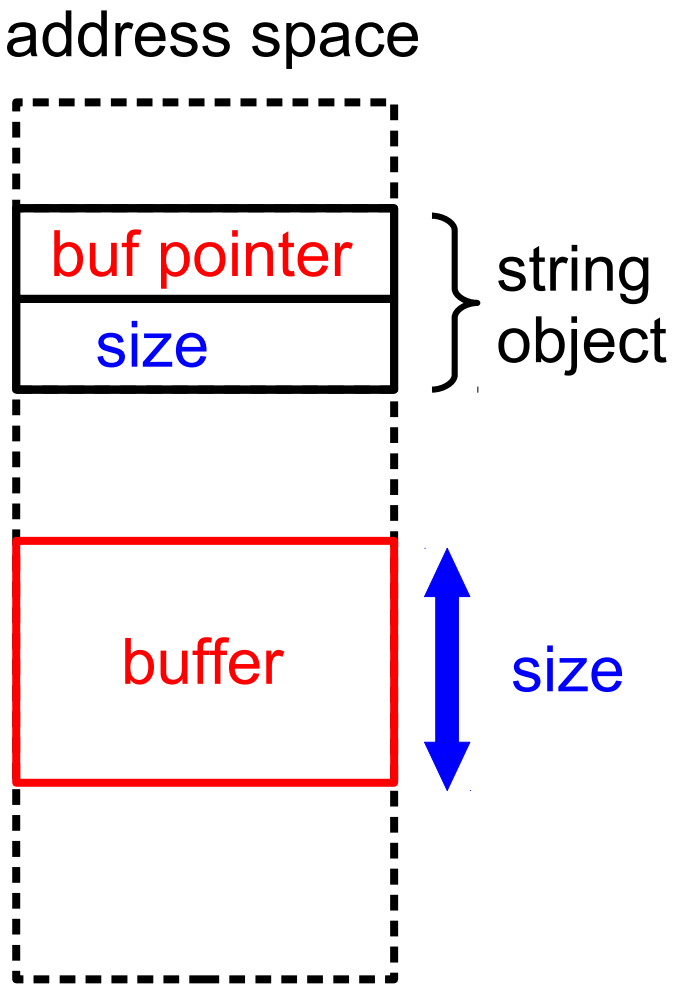
  → leak information into context of script-engine

- Attacker can abuse script engines (i.e., JavaScript)

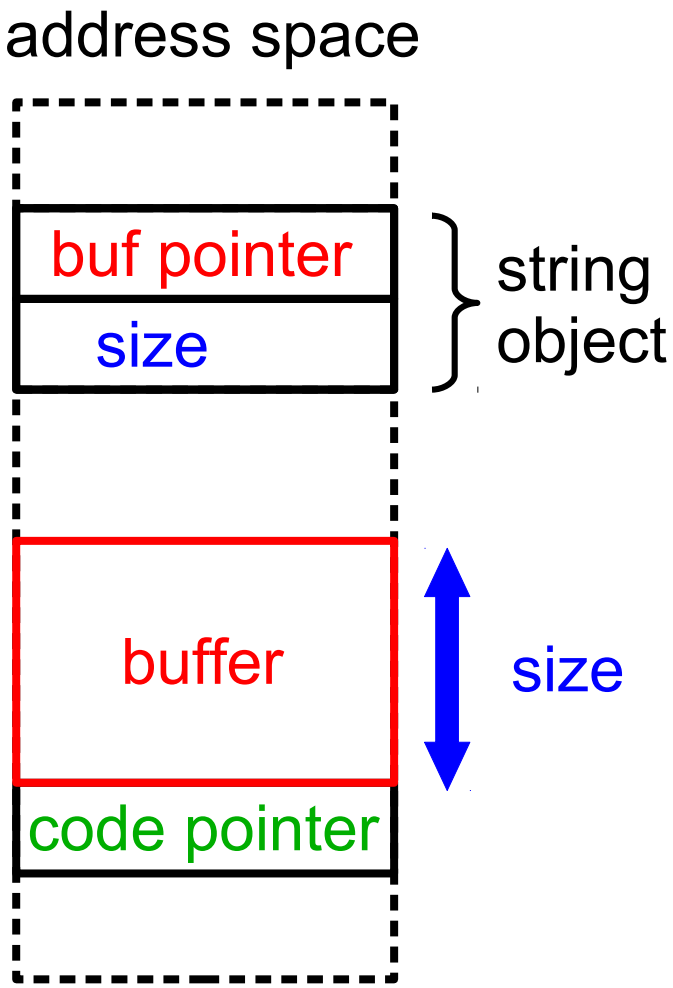
  → Manipulation of internal script-engine objects
  - Pointer manipulation
  - Size field manipulation

  → very powerful

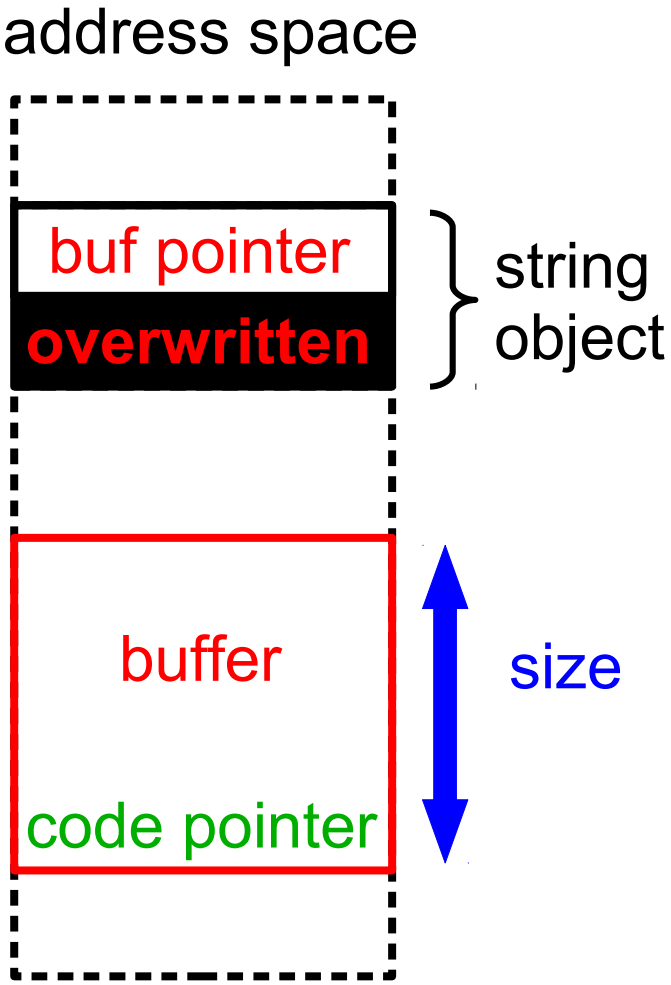
Manipulation of internal script-engine objects



Manipulation of internal script-engine objects



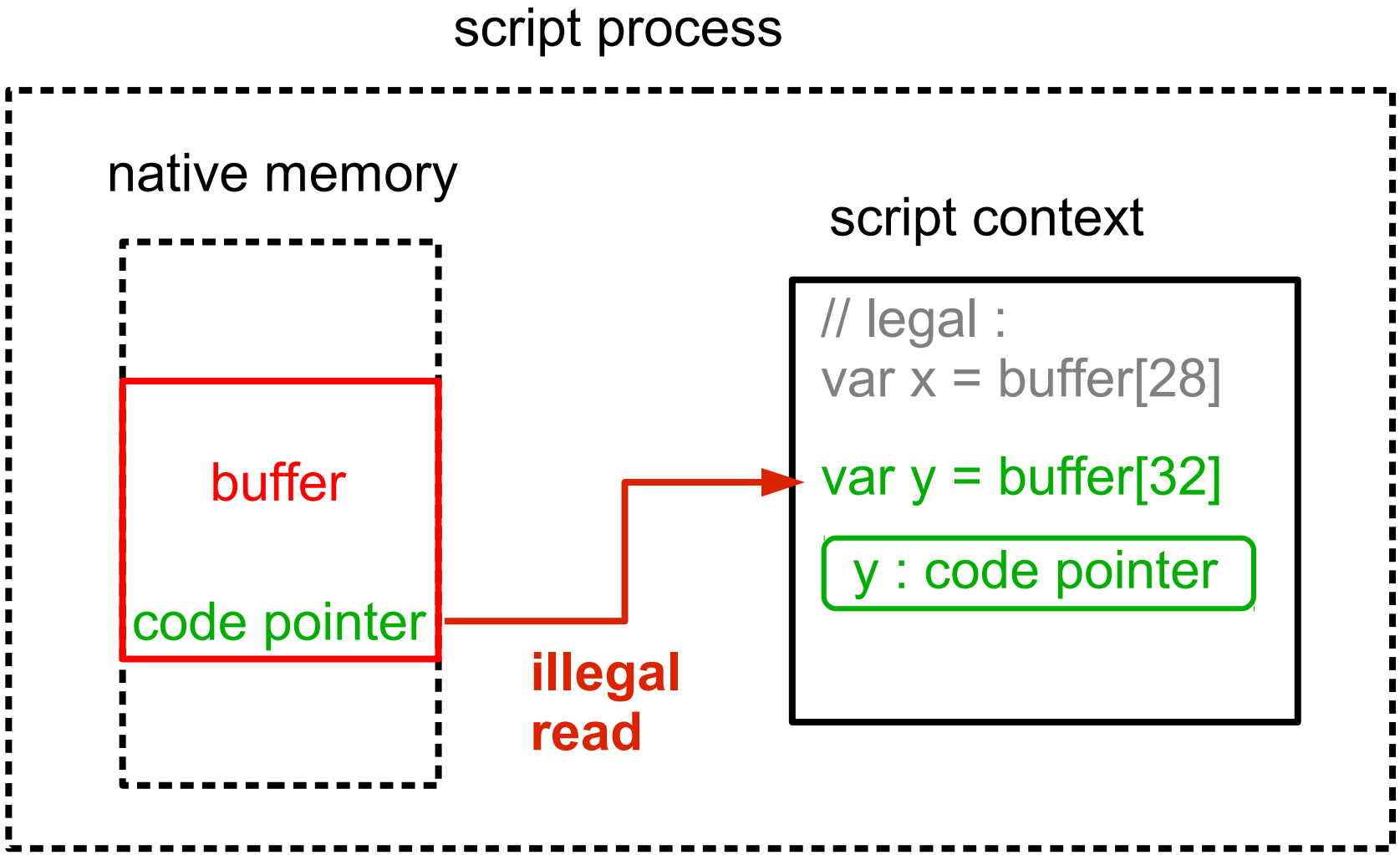
Manipulation of internal script-engine objects



- Attacker overwrites length field of string object

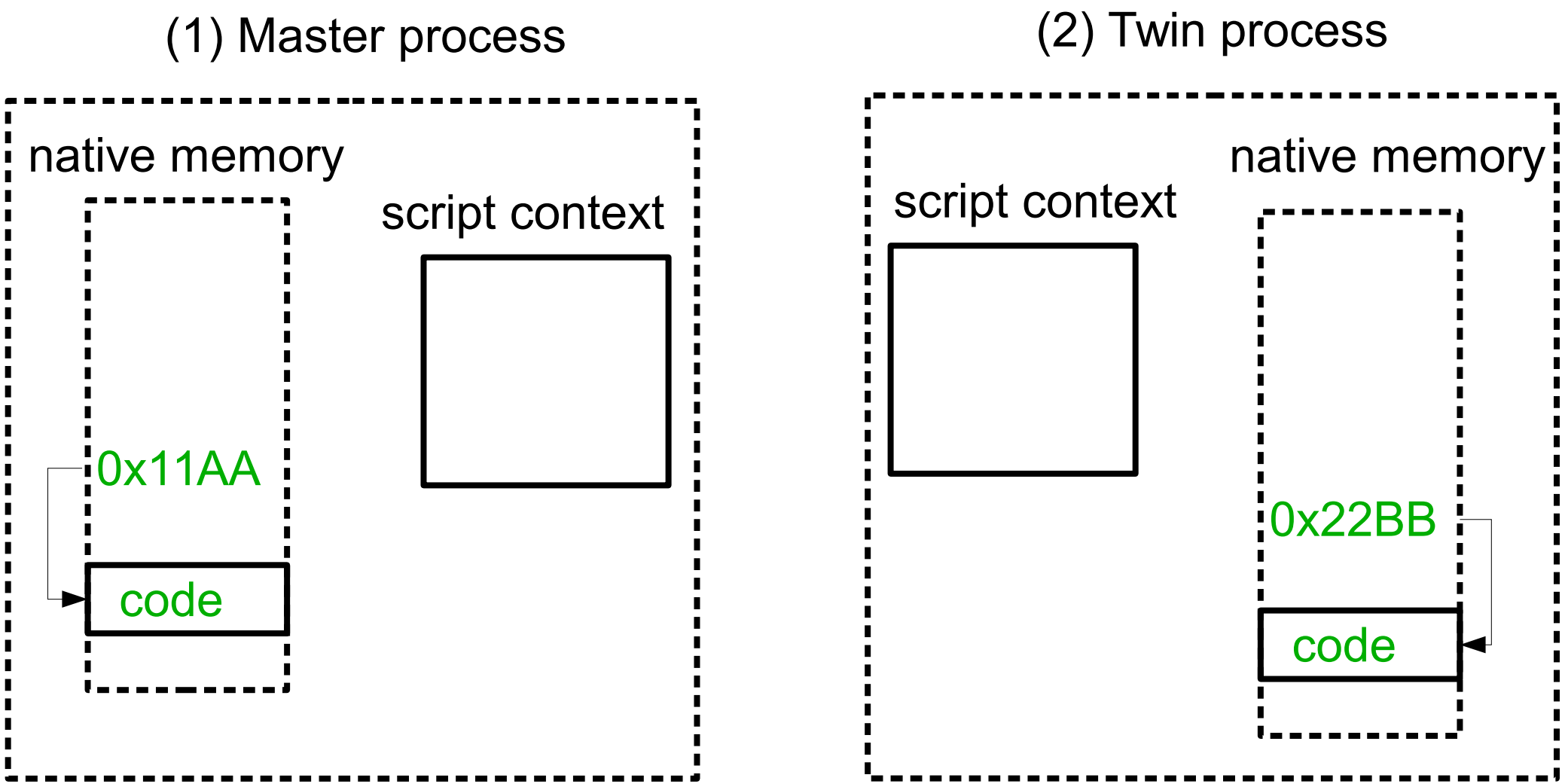
- Use string-object methods to leak code pointer into script context

Information leak into script context



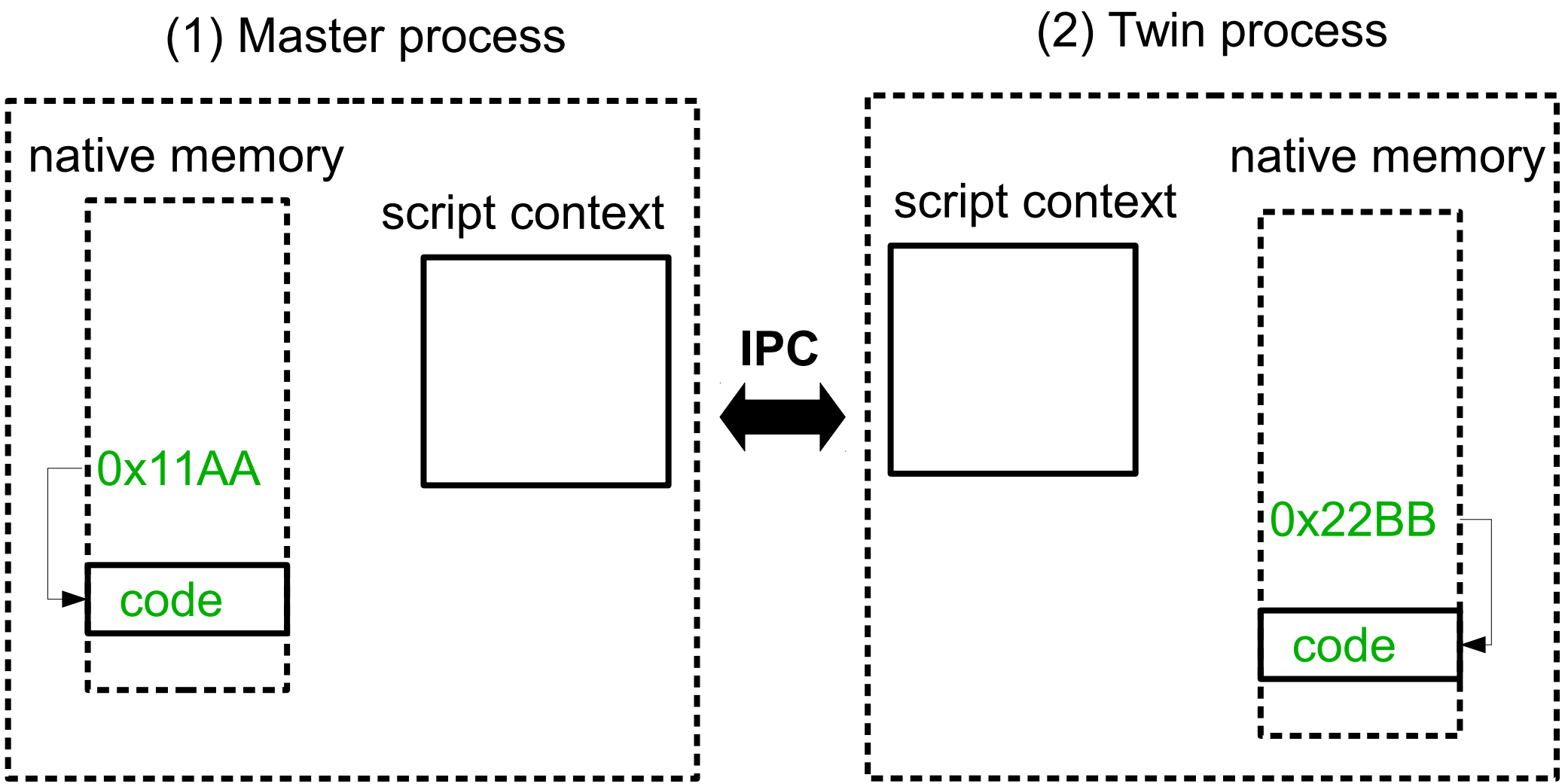
Design

Information-leak detection



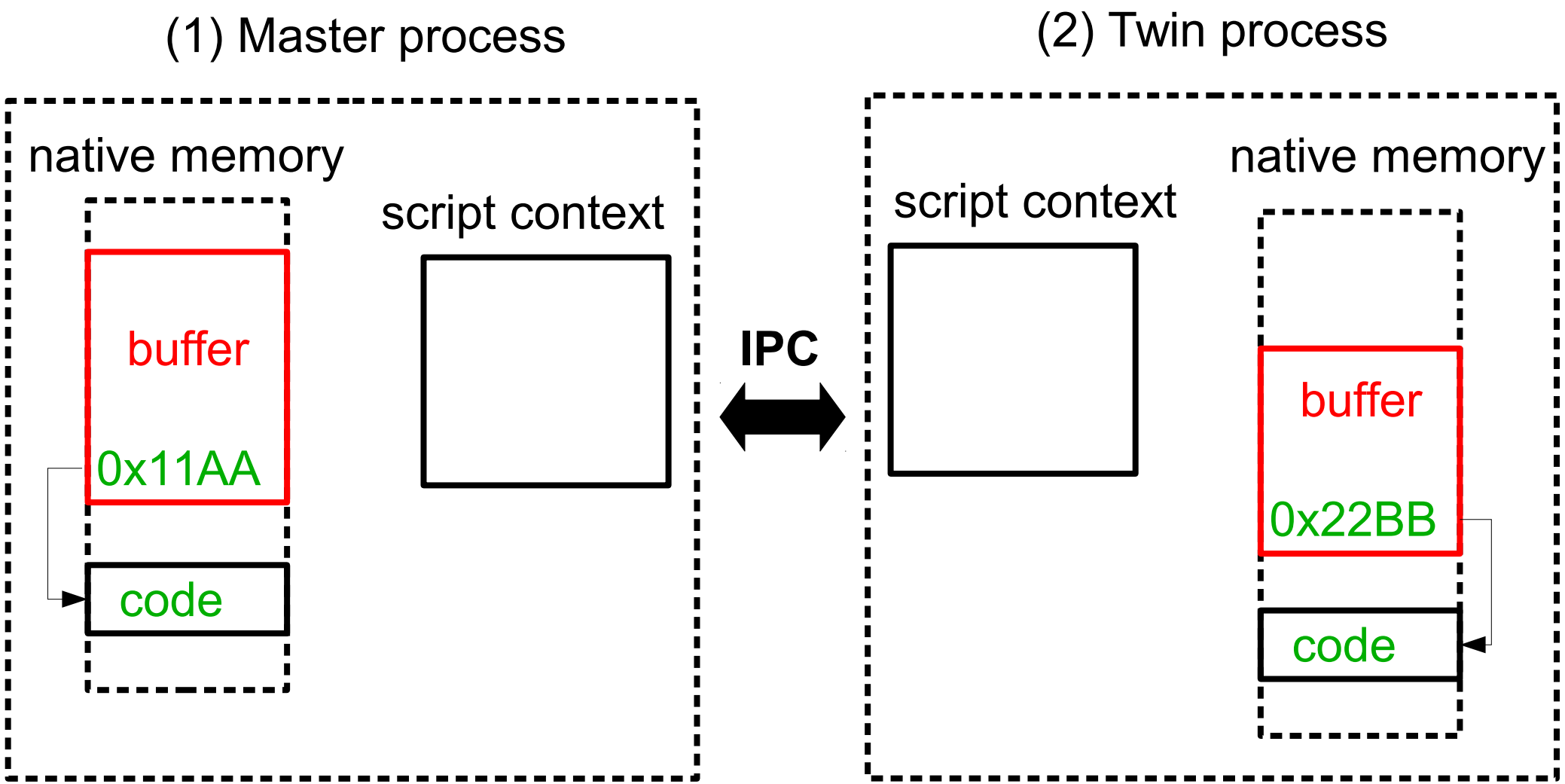
Main Concept

Information-leak detection

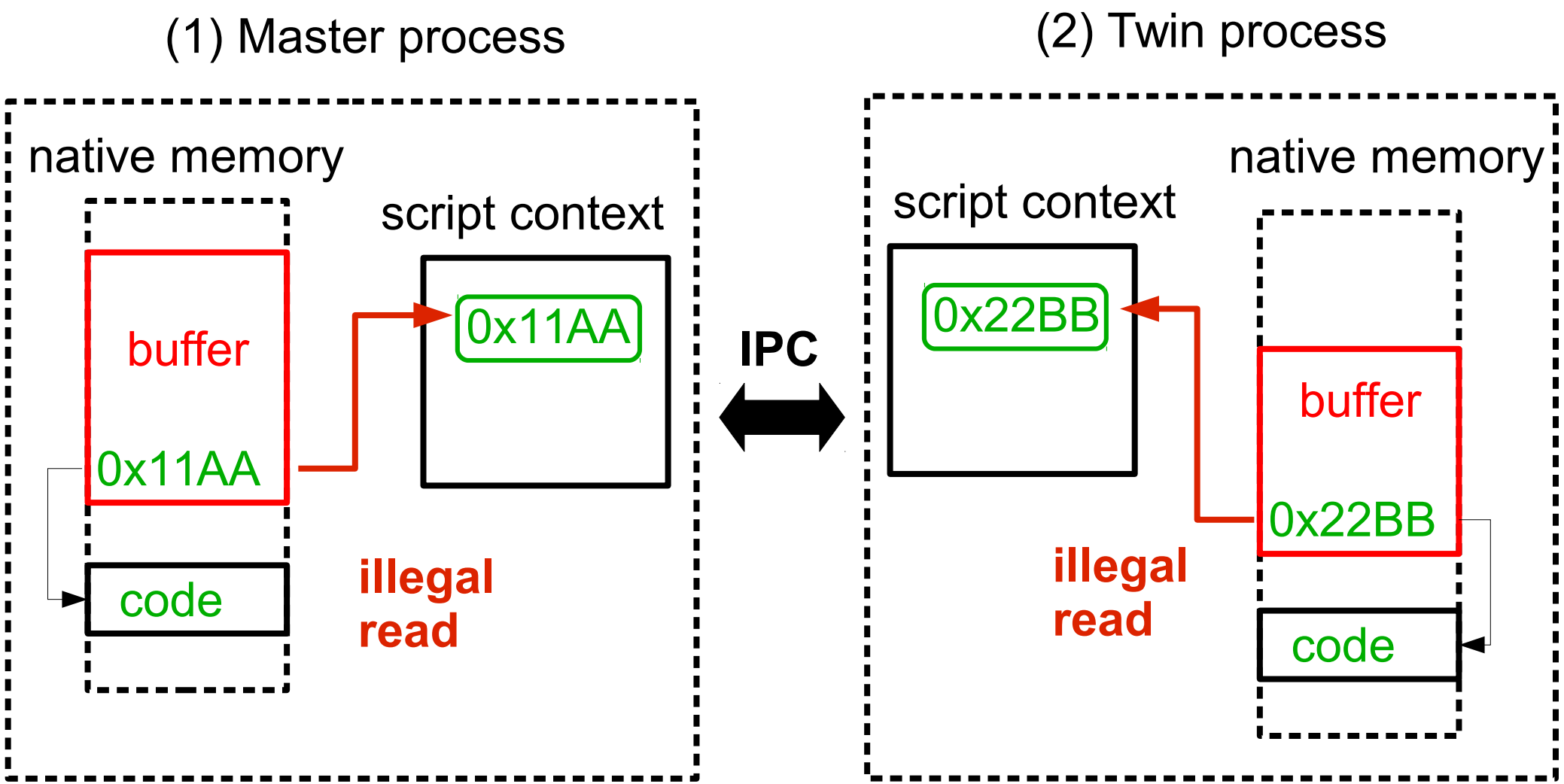


Main Concept

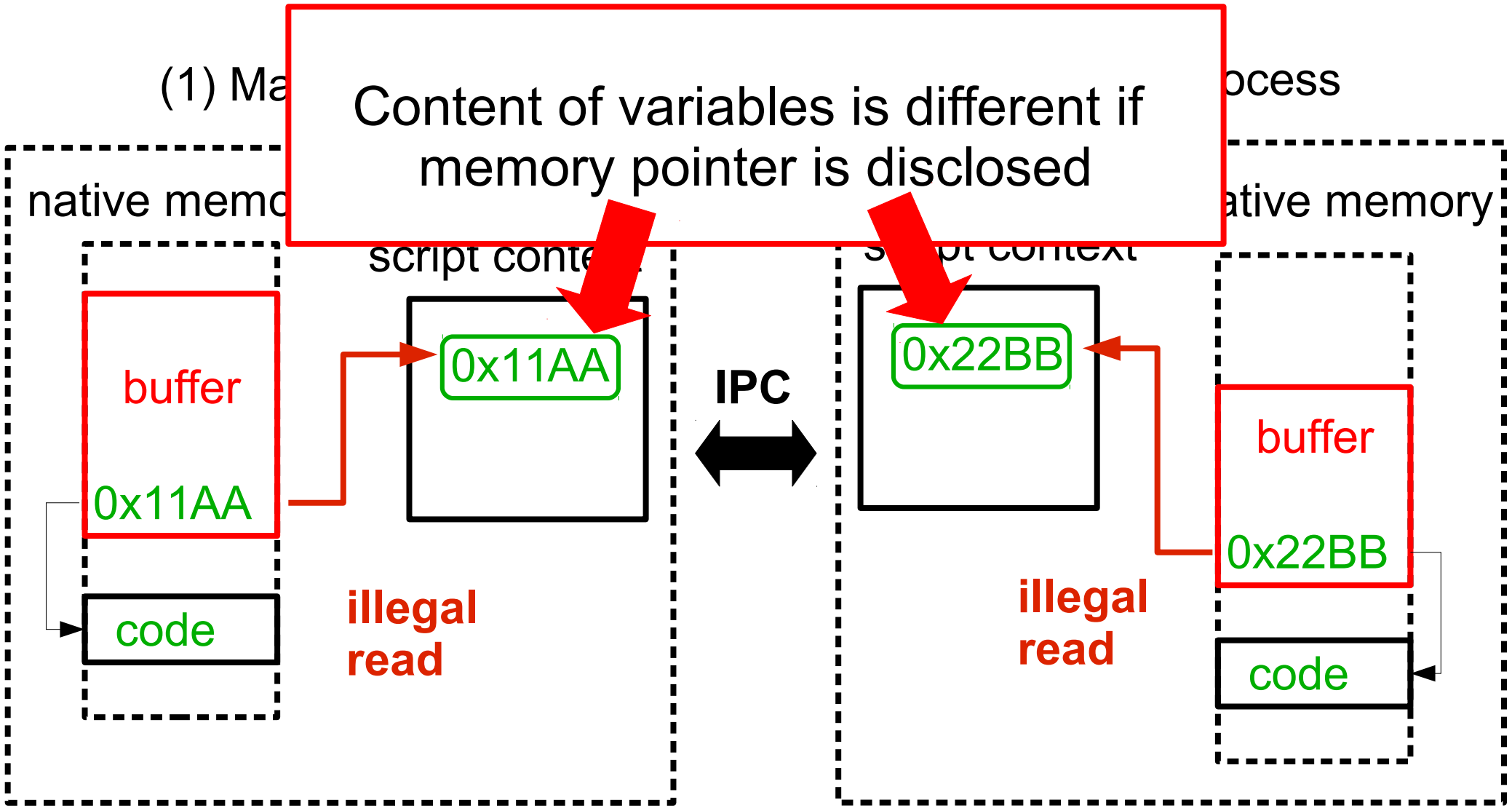
Information-leak detection



Information-leak detection



Information-leak detection



Main Concept

- Execute two instances of script process (e.g., web browser)
- Enforce different address space layout in both instances
- Synchronize execution of both instances and execute same web data
- Check content of script variables in both instances as they are assigned

→ A different content of the same variable in both instances indicates an ongoing memory disclosure

Implementation

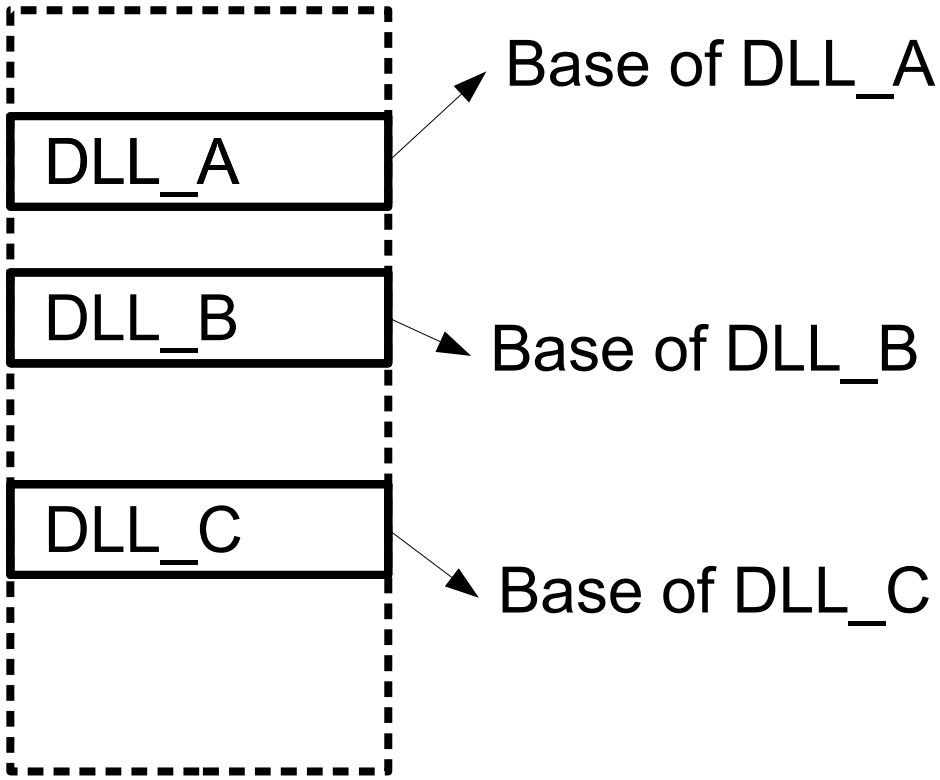
Re-Randomization

- Mapped images (e.g., DLLs) have equal base addresses across processes (Windows) → Not ideal for our approach

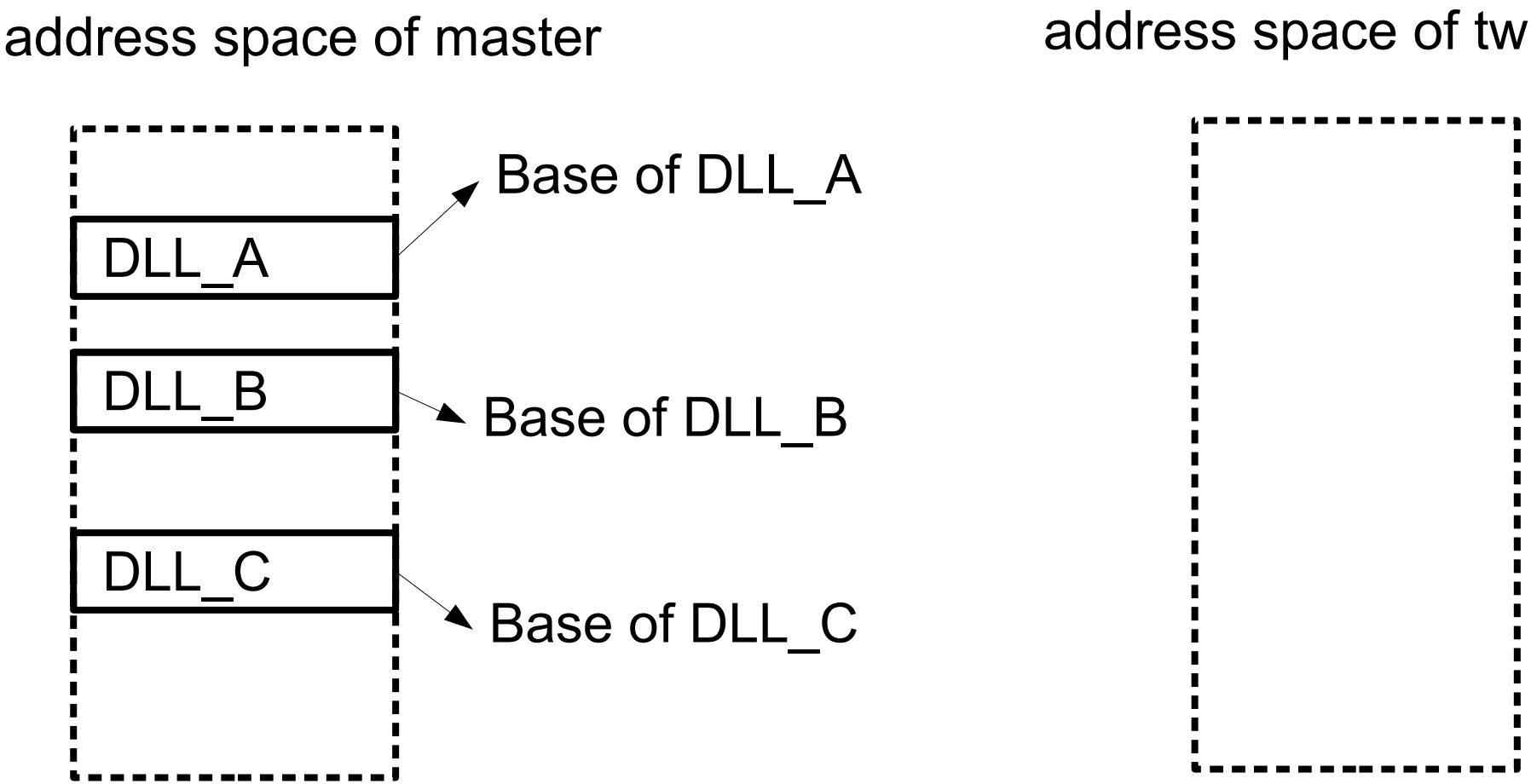
(1) Master: retrieve base addresses of mapped images
(2) Twin: occupy base addresses

Retrieve base addresses of DLLs in master process

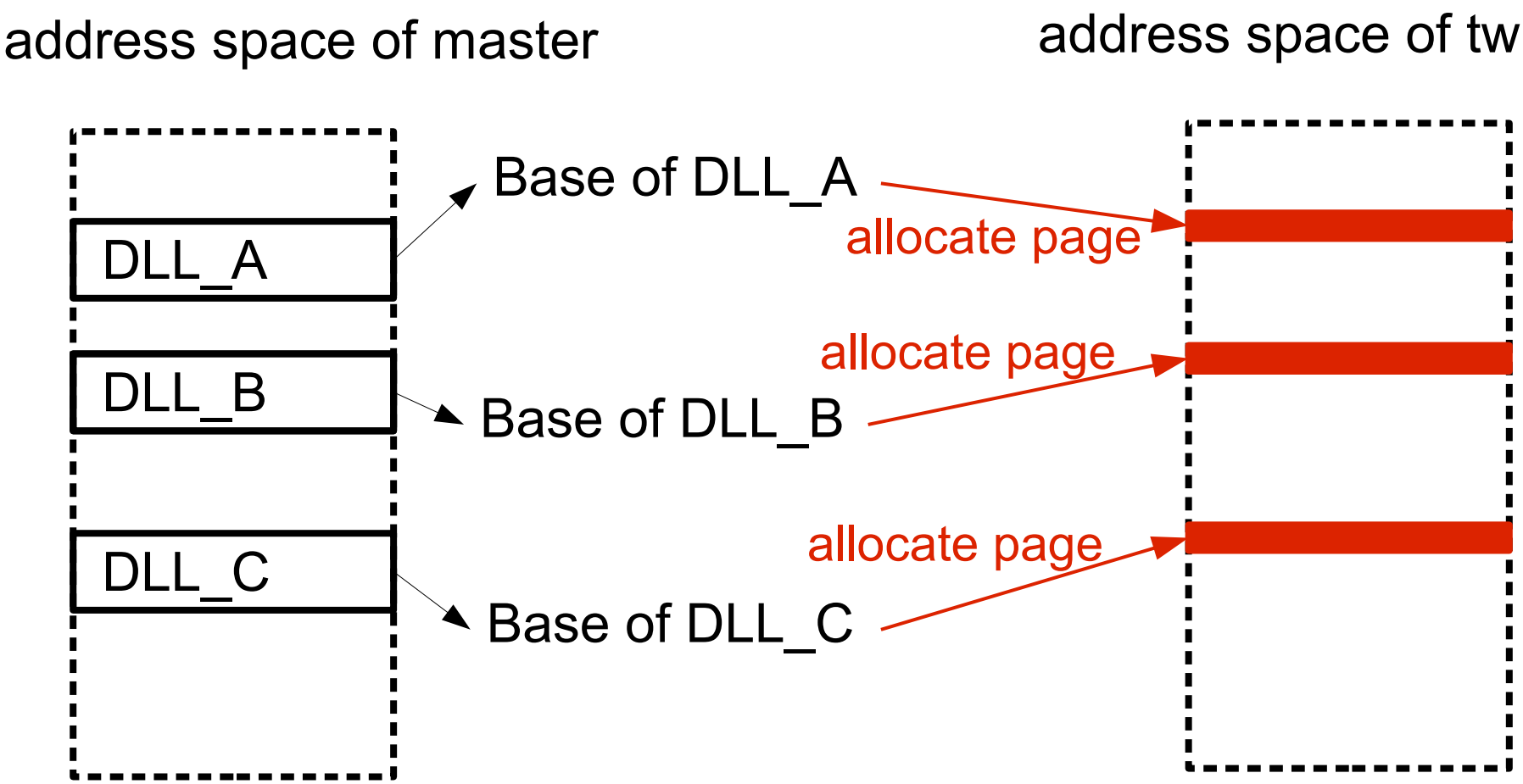
address space of master



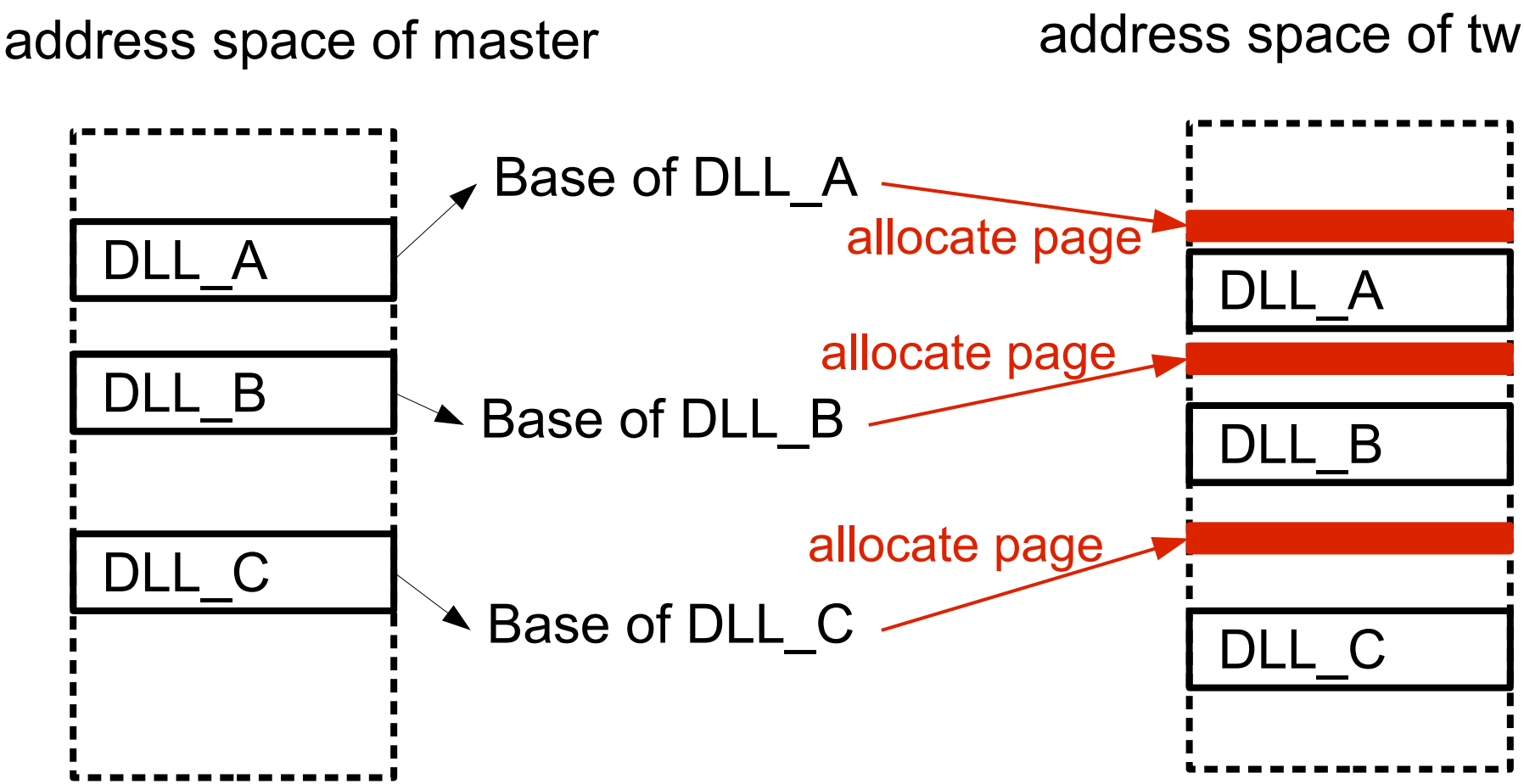
Enforce different address space in twin process



Enforce different address space in twin process



Enforce different address space in twin process



Re-Randomization

- Mapped images (e.g., DLLs) have equal base addresses across processes (Windows) → Not ideal for our approach

  (1) Master: retrieve base addresses of mapped images
  (2) Twin: occupy base addresses

  → Loader maps DLLs to different base addresses in twin process
  – Specific DLLs require special handling

- Stack and heap memory regions have already different base addresses per process due to ASLR

Synchronization

- Instrument native functions
- Instrument bytecode handlers of script interpreter
  → e.g., *call*, *return*, *conversion* bytecode handler

→ Synchronization and checking points

  - master drives execution
  - twin follows execution
  - comparison of data flows between master and twin (script context ←→ native context)

  → fine-grained

Information-Leak Detection

→ Compare script function return values
return bytecode: native context → script context

→ Compare script function arguments
call bytecode: script context → native context

Information-Leak Detection

→ Compare script function return values
return bytecode: native context → script context

→ Compare script function arguments
call bytecode: script context → native context

Consistent data is required:

- Proxy relays web data received by master to the twin
  → Ensure web data is identical

- Entropy elimination (e.g., Math.random())
  → Pass return value from master to twin

Evaluation

Evaluation

Successful information-leak detection

Uninitialized variables:
- JPEG parsing bug (CVE-2014-6355)
- TIFF processing bug (CVE-2015-0061)

→ leak stack addresses

Typed array pointer and size field manipulation with CVE-2014-0322

→ leak vtable pointer

Evaluation

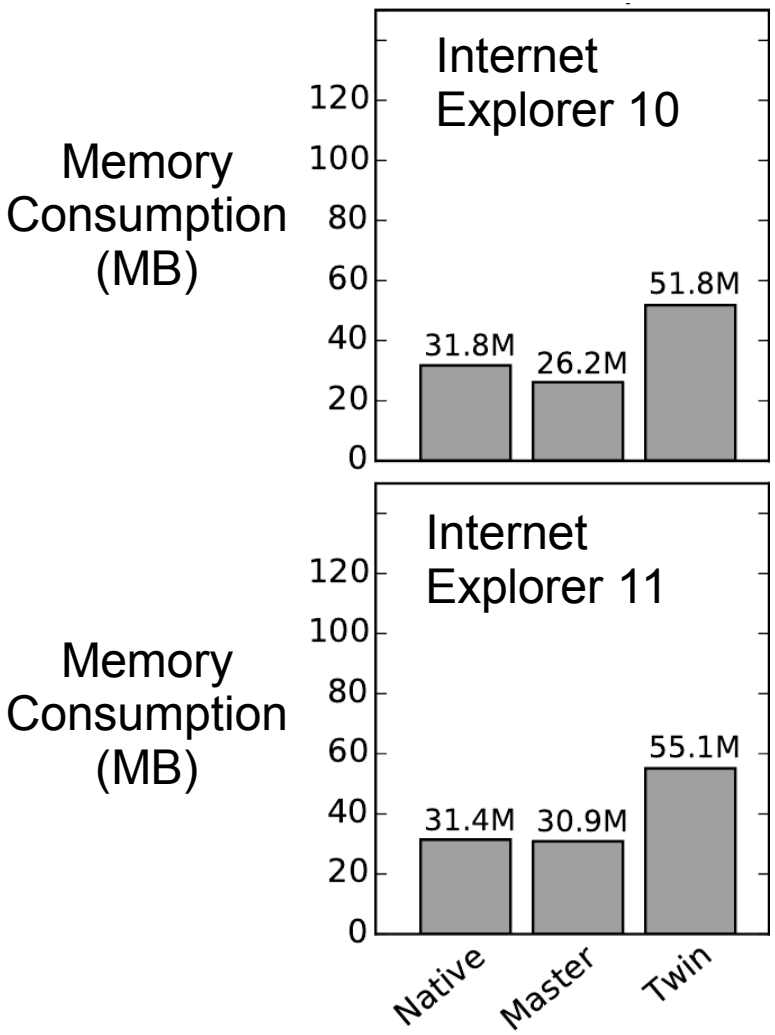
Program startup overhead

	Native	*Detile*	Slowdown
Internet Explorer 10 (new tab)	0.92 s	2.07 s	1.3 x
Internet Explorer 11 (new tab)	0.52 s	1.31 s	1.5 x

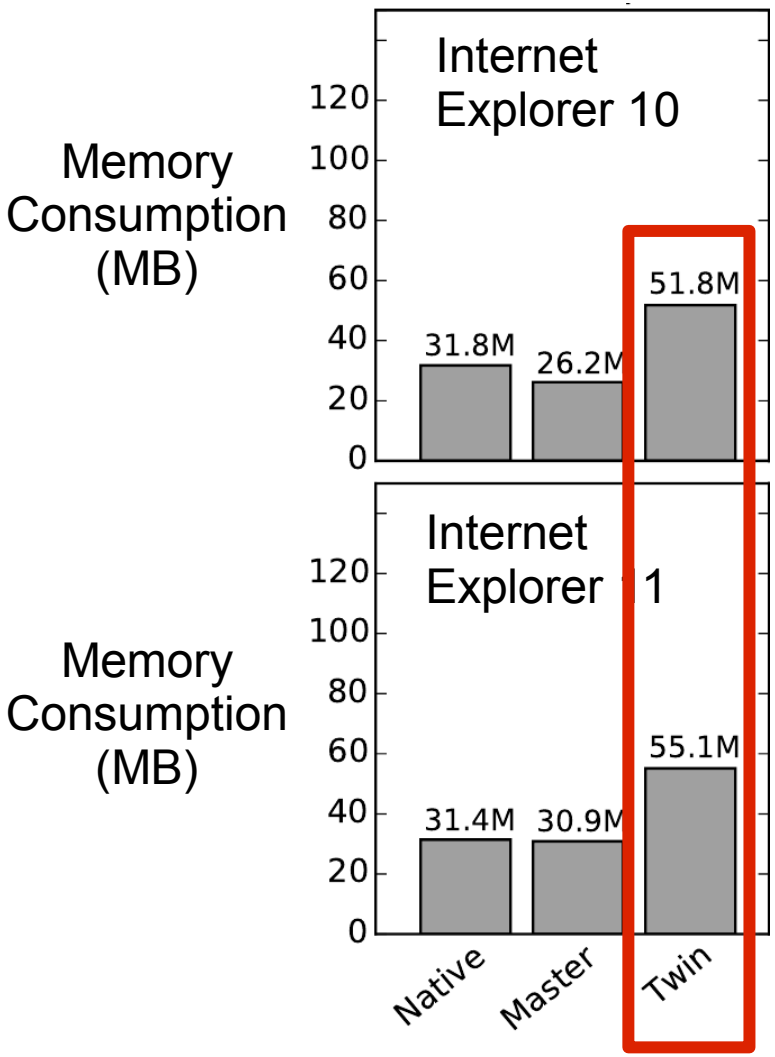
- Each new tab in Internet Explorer is a new process
  - New tab process becomes the master process
  - Additional twin process per master process

→ Increased startup time

Memory overhead



Memory overhead



Each twin process has private DLL copies

→ Physical memory is not shared across DLLs in processes

→ Additional memory consumption

Evaluation

Performance overhead

Script execution time in Internet Explorer 11 (ms)

Web page	google.com	facebook.com	youtube.com	yahoo.com	baidu.com	wikipedia.org	twitter.com	qq.com	taobao.com	linkedin.com	amazon.com	live.com	google.co.in	sina.com.cn	hao123.com
Native	425	774	1196	3674	1108	472	599	2405	645	439	958	254	483	3360	373
DETILE	482	961	1519	4722	1339	513	623	2724	824	517	1210	275	517	4269	379
Overhead (%)	13.4	24.1	27	28.5	20.8	8.6	4	13.2	27.7	17.7	26.3	8.2	7	27	1.6

→ On average: 17 % overhead

Conclusion

Conclusion

- Information leaks are used as fundamendal step in modern memory corruption exploits
- Dual execution/synchronization of script-engine processes can detect information leaks
- Each script engine has to be handled separately

  → Detailed knowledge of engine's internals necessary
  → Manual, time consuming effort
  → Hard for binary-only code

- Induces measurable overhead

Q & A

References

[1] Fermin J. Serna. **The Info Leak Era on Software Exploitation**. *Black Hat USA, 2012*.

[2] Hiser et al. **ILR: Where'd My Gadgets Go?** *Security & Privacy, 2012*.

[3] Snow et al. **Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization**. *Security & Privacy, 2013*.

[4] Tang et al. Heisenbyte: **Thwarting Memory Disclosure Attacks Using Destructive Code Reads**. *CCS, 2015*.

[5] Snow et al. **Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks**. *Security & Privacy, 2016*.

[6] Onarlioglu et al. **G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries**. *ACSAC, 2010*.

[7] Athanasakis et al. **The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines**. *NDSS, 2015*.