

# Picking up the trash

## Exploiting generational GC for memory analysis

**Adam Pridgen**<sup>1</sup>   Simson L. Garfinkel<sup>2</sup>   Dan S. Wallach<sup>1</sup>

<sup>1</sup>Rice University, Houston, TX, USA

<sup>2</sup>George Mason University, Fairfax, VA, USA

Digital Forensics Research Workshop, 2017



- Java runtime uses automatic memory management
- Data lifetimes are not controllable
- Data cannot be explicitly destroyed
- Multiple copies are typically created

- Cross-platform malware uses managed runtimes
- Threat actors also exploit vulnerable applications
- Managed runtimes retain many artifacts

- Can object meta-data be exploited for investigations?
- What kind of information is obtainable?
- Can viable timelines be created?
- Can the approach be generalized to other runtimes?

- 1 Introduction
- 2 Managed Memory Overview
- 3 Approach
- 4 Evaluation
- 5 Conclusions

## Generational GC Heap Overview

- Tracing GC: Looking for *live* objects from a set of roots
- Heap engineered for expected object life-time
- GC promotes objects from one heap to the next one
  - **Eden Space** (short lived) → **Survivor Space**
  - **Survivor Space** → **Tenure Space** (long lived)

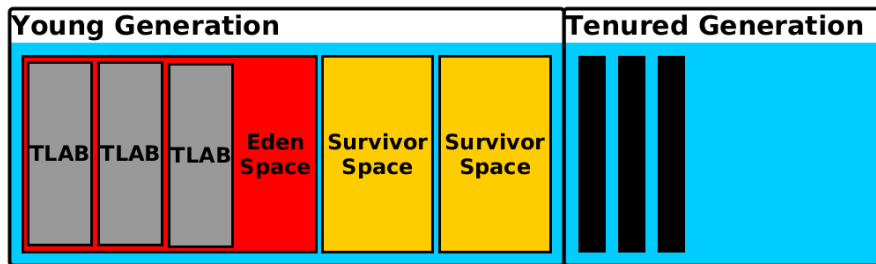


Figure: Typical generational heap layout.

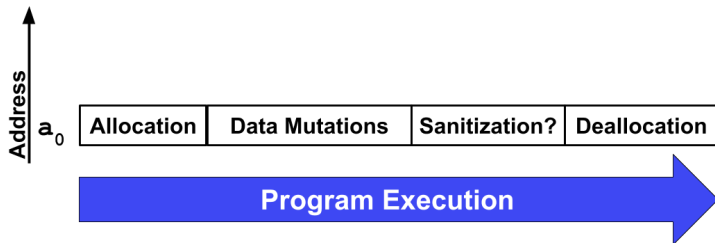


Figure: Example data lifetime in unmanaged memory.

## Managed Data Lifetime Overview

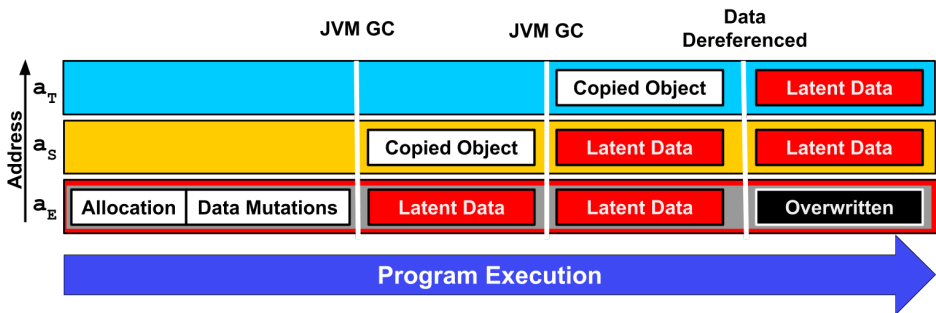


Figure: Example data lifetime in managed memory.



# Why is data being retained?

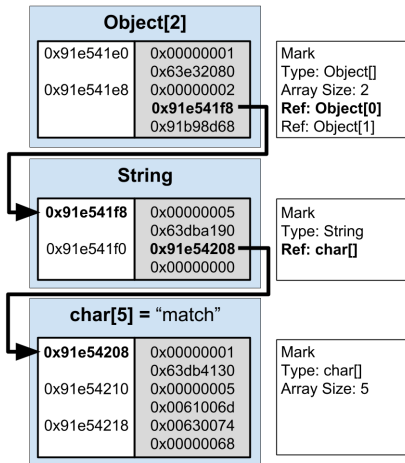


Figure: String[2] on the heap.

## Why is data being retained? (2)

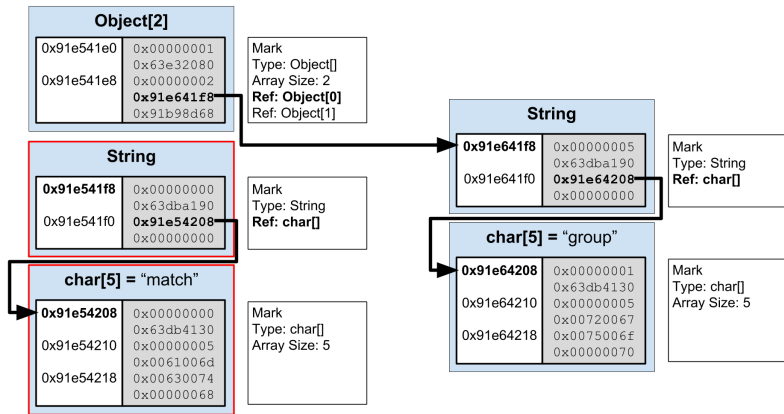
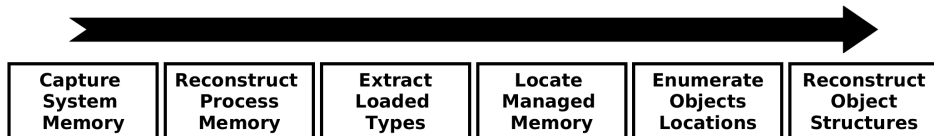
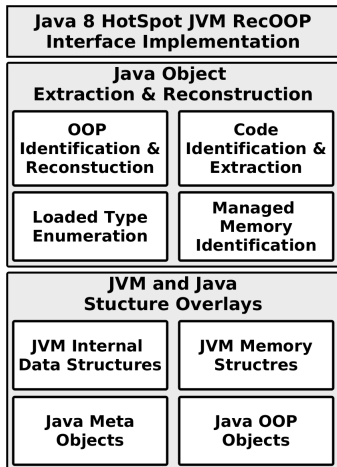


Figure: `String[0]` is reassigned but the old value remains.



# Recovering OOP Framework (2)

- Focuses recovery from x86 architecture
- Uses a minimal set of structure overlays
- Compatible with Linux and Windows OS



- Identify structures revealing loaded types
  - `SystemDictionary`: loaded classes
  - `SymbolTable`: loaded symbols
  - `StringTable`: constants or long-lived strings
- Mine structures for the loaded data structures

# Extract Loaded Types (2)





- Look for invariant values
- Walk the hash tables
- Use constraints to control recovery

**Table:** The regular expression “`space.*used`” used in conjunction with `ffastrings` to determine the eden, survivor, and tenure generation spaces. Note `[...]` signifies omitted message content.

	GC Log Message	
Generational Space	Start and End of the Space	
<b>eden space</b>	<code>[...] used</code>	<code>[0xa4800000, [...] 0xa4c50000)</code>
<b>from space</b>	<code>[...] used</code>	<code>[0xa4c50000, [...] 0xa4cd0000)</code>
<b>to space</b>	<code>[...] used</code>	<code>[0xa4cd0000, [...] 0xa4d50000)</code>
<b>the space</b>	<code>[...] used</code>	<code>[0xa9d50000, [...] 0xaa800000)</code>

## Locating Managed Memory with Pointers

**Table:** Java object distribution in managed process memory (e.g. eden, survivor, and tenure spaces).

Address Range	Type Pointers	Unique Pointers	Pointer Occurrences Per Page (Y-axis: 0-64)
0xa47ff000-0xa4c0f000	13261	266	
0xa4c50000-0xa4c92000	129	28	
0xa4cd0000-0xa4d50000	1121	79	
0xa9d50000-0xaa000000	28810	661	



# Enumerate and Extract Objects

- Scan managed heap for known types
- Parse the object based on the report type
- Lift values for the object's fields

**Java Threads****Sockets****Process Builders****Native Buffers****Streams****Child Processes****JAR Files****Buffers****JAR Entries**

- Created software similar to a malware implant
- Used a script of common threat actor activities
- Took memory snapshots after each activity
- Analyzed the snapshots using RecOOP

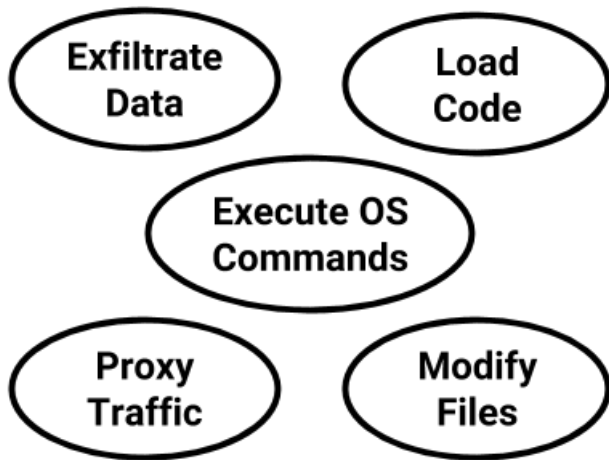


Figure: Overview of the malware functionality for experiment.

## Reconstructing Socket Connections

Object Address	Remote Connection	In/Out		Data (Up to 30 Bytes)
0x91c779b8	10.18.120.18	48002	⇒	Do something evil-48002!
0x91c7ead0	10.18.120.18	48003	⇒	Do something evil-48003!
0x91c85b70	10.18.120.18	48002	⇐	s3cr3t_d4t3_48002-00000000s3cr
0x91c938d8	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDItMDAw
0x91c980d0	10.18.120.18	48003	⇐	s3cr3t_d4t3_48003-00000000s3cr
0x91ca5cb8	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDMtMDAw
0x91cbfef0	10.18.120.18	48004	⇒	Do something evil-48004!
0x91cc7008	10.18.120.18	48005	⇒	Do something evil-48005!
0x91ccdee8	10.18.120.18	48004	⇐	s3cr3t_d4t3_48004-00000000s3cr
0x91cdbad0	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDQtMDAw
0x91ce02c8	10.18.120.18	48005	⇐	s3cr3t_d4t3_48005-00000000s3cr
0x91cedeb0	172.16.124.15	58860	⇒	czNjcjN0X2Q0dDNfNDgwMDUtMDAw

**Table:** This table shows a sampling of the processes started by the Java program and the `stdout` buffer at `t=21`.

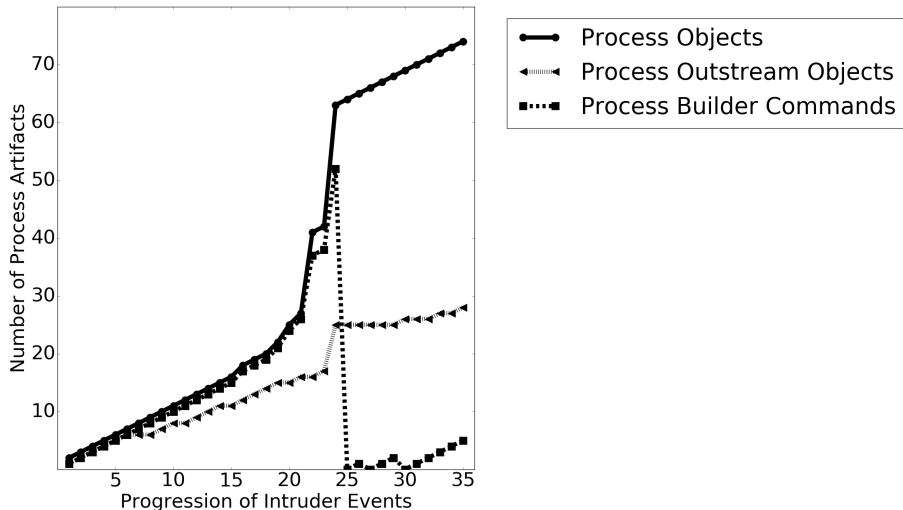
Address	PID	Buffered Data
0x91dff7e0	1242	#\n# This file MUST be edited w
0x91e1c7e8	1245	Linux java-workx32-00 3.19.0-1
0x91e3b0e0	1248	java adm cdrom sudo dip plugde
0x91e4a6e8	1250	root:x:0:0:root:/root:/bin/bas
0x91eb1390	1252	root!:16678:0:99999:7:::\ndaem
0x91f66708	1275	\nStarting Nmap 6.47 ( http://n
0x91ff7ed0	1301	history   grep pg\n history   gr
0x92014f30	1307	ifconfig\nsudo add-apt-reposito
0x920626d8	1322	adding: home/java/.ssh/ (sto

**Table:** This table shows a selected set of method call data extracted from the JVM. **Unused functions** have a `null` counter value.

Address	Calls	Method Name
0x63fdb6f8	256	Loader getLoaderInstance(...)
0x63fdb908	73	byte[] b64Decode(...)
0x63fdce98	256	integer sendSocketData(...)
0x63fdd718	256	void stdout(...)
0x63fdd850	256	void logEvent(...)
0x63fddb8	73	integer getPid(...)
0x63fddd50	73	integer startProcess(...)
0x63fddf68	256	java.lang.String readProcessStdout(...)
0x63fdd9e8	1	void main(...)
0x63fdb670	1	void start(...)

# Evaluating Event Reconstruction: Process Objects

## RICE





# Conclusions

- Memory analysis can recover these artifacts
- Meta-data and data locality help with reconstruction
- GC memory allocation eases timeline creation
- This approach applies to other Generational GCs

