Interpreter Exploitation

Pointer Inference and JIT Spraying

dion@semantiscope.com

Interpreter Exploitation

Pointer Inference and JIT Spraying

or How I Learned to Stop Worrying and Pop the Vista Box

dion@semantiscope.com

The Promise (or why to stick around)

I will explain why, despite the many new exploit mitigations available in Vista/Win7, you still aren't safe surfing the web on Vista with IE8 (with Flash Player, but who doesn't like YouTube?).

The Story so far...

Our protagonist's name is Leon Plazakis

The Story so far...

He has recently been hired as a “Security Analyst” for Defense R' Us. They contract to the Highlandia Government.

The Story so far...

His latest top secret assignment is to design an exploit targeting Charlie Root.

Charlie is secretary to the King of Midlandia (they suck).

Intelligence

Here is what we know (Leon's gang of ninja spies told him this):

Intelligence

Charlie has a laptop running XP SP3. He browses with IE7 and has both Adobe Flash Player and Adobe Reader installed.

Intelligence

Charlie's IT team is not completely bad. He is usually up to date on patches.

Intelligence

He isn't clueless enough to run an executable attached to an e-mail. His security staff has even convinced him clicking on links in e-mails is bad.

Well... usually...

Intelligence

He does have one weakness – he loves the Flash game site “Super Innocent Games.” SIG is a community run site.

Leon's hacker senses are tingling…

The Plan

Leon is “leet”. He has 368 fuzzers.

So, step 1 – Find 0-day in IE7, Reader, or Flash.

The Plan

Step 2 – Create/steal a Flash game to put on the site. He will incorporate the 0-day and check for Charlie's username before triggering.

The Plan

Step 3 – E-mail the link from Charlie's old college roommate via spoofing or hacking or rubber hose. Charlie always gets new SIG game suggestions from him.

The Plan

Step 4 – Profit!

Raise! Bonus! Leetness!
(Leon is thinking big)

Plan A – In Action!

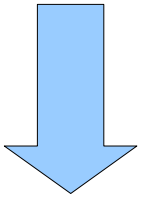
After some binary reversing and a few cups of coffee, Leon finds an IE 0-day (it works on IE6-8!)

Leon calls it the Faurora bug (after his first girlfriend)

Plan A – In Action!

Here is his proof of concept:

```
window.sekretMethod(0x41414141);
```

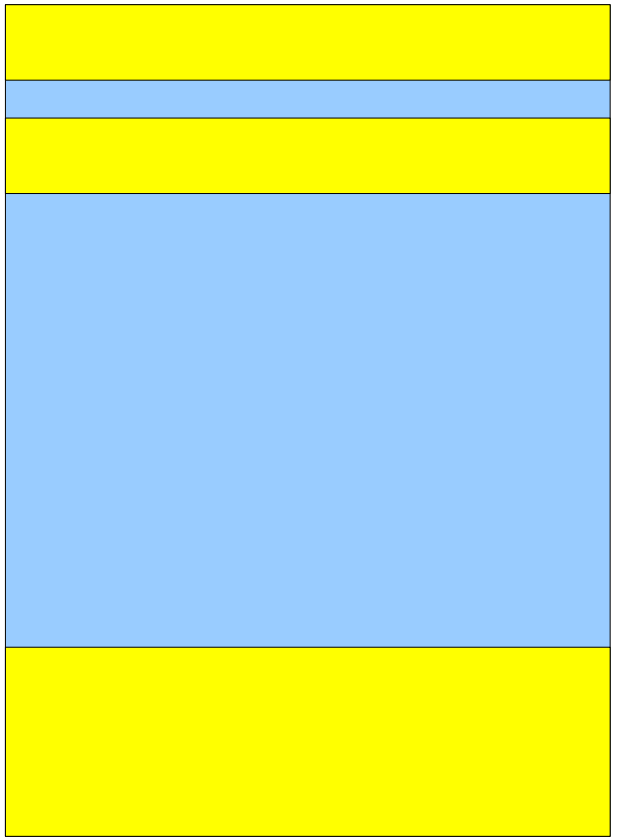


```
call eax ; eax = 41414141
```

Plan A – In Action!

Leon is now very excited – a simple heap spray and Leon will have code execution.

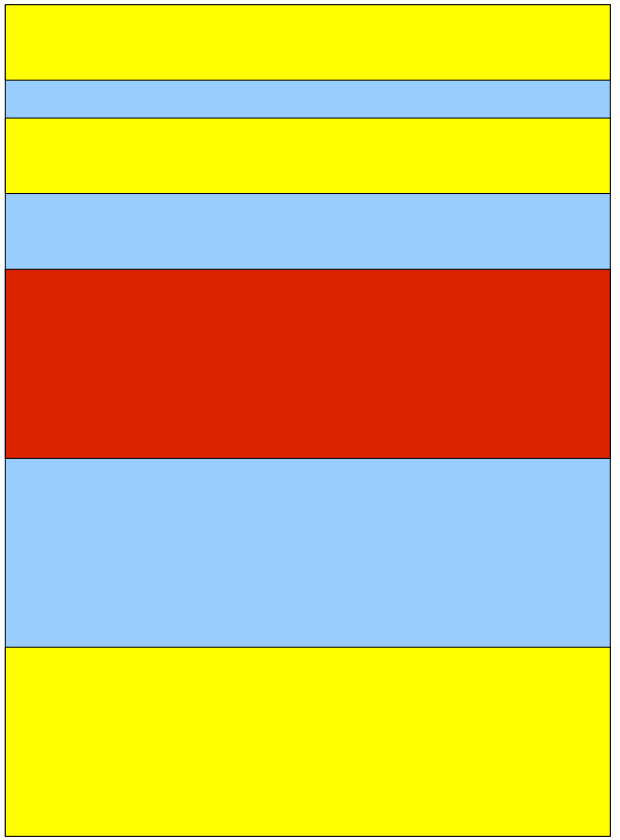
Aside: Heap Spray



Do this a bunch of times (easy to do in Javascript):

1. Make a big (really) NOPsled ending in shellcode.

2. Hold on to a reference to this string.



Plan A – In Trouble!

Uh oh. Leon's luck has run out...

He was adding the last particle effect to his Flash Gordon game (the host for his exploit) when a co-worker broke the news.

Plan A – In Trouble!

Lowlandia attacked Midlandia

Midlandia security is now juiced up. Charlie will be browsing with IE8. His laptop has been upgraded to Vista.

Plan A – Out the window!

Leon is leet. He has read about DEP and ASLR.

Leon considers a farming job.

Aside: DEP

DEP is Data Execution Prevention

This maintains a strict separation between code and data.

Aside: DEP

Think of it this way:

Microsoft hates Reese's cups.

“You've got your code in my data section!”

Aside: DEP

How does this help?

Hint: Code section aren't writeable.

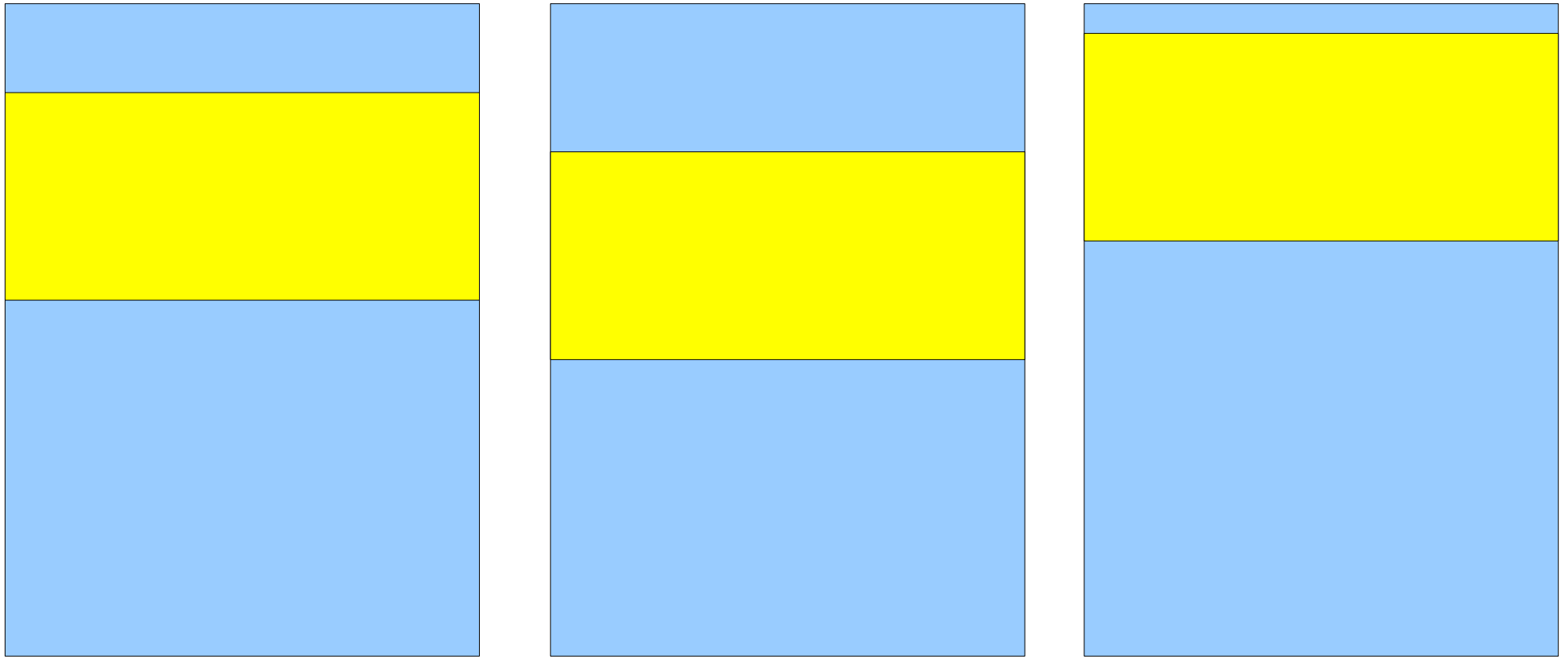
How do you get shellcode into a code section?

Aside: ASLR

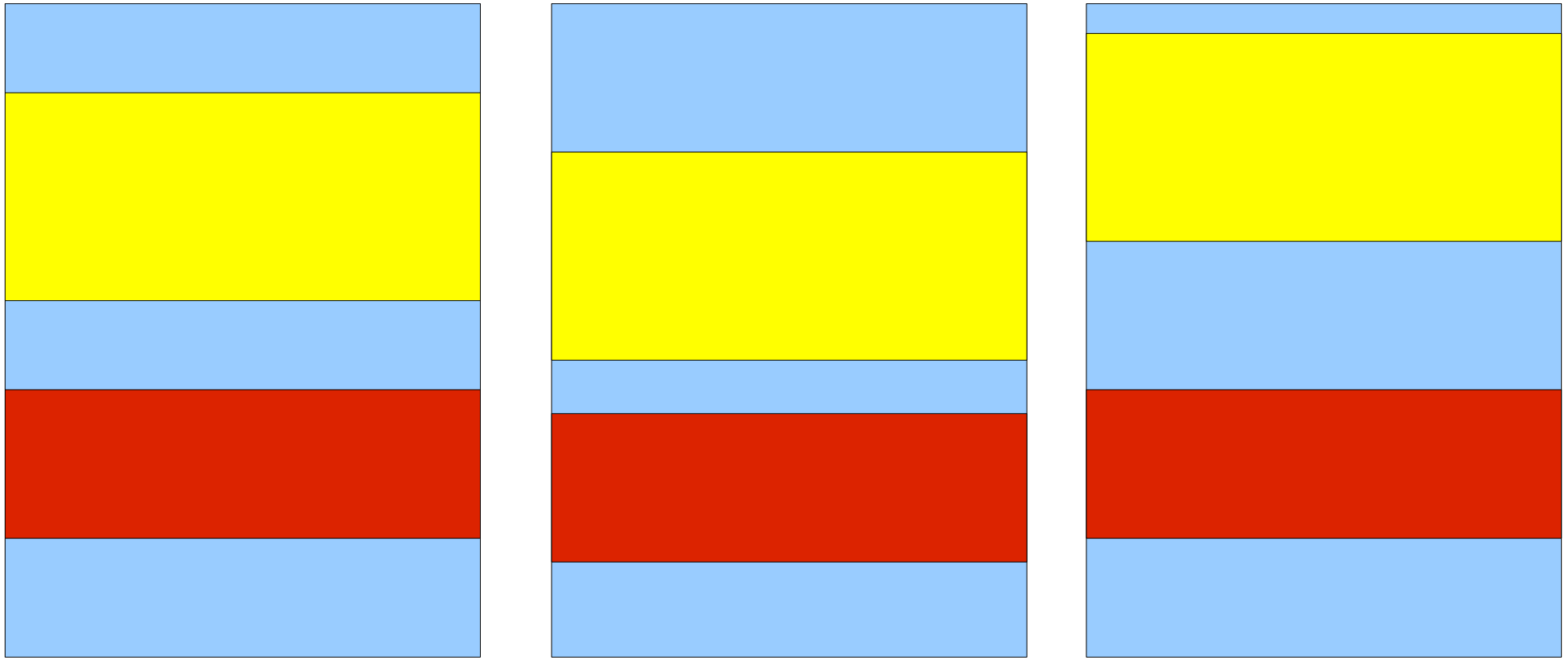
Address Space Layout Randomization

Avoid determinism in the locations of:
Loaded Images (.exe, .dll)
Heap (dynamically allocated memory)
Stack

Aside: ASLR



Aside: ASLR



Aside: DEP + ASLR

When the two combine...

Hackers everywhere shed a tear.

Back to Leon...

Leon decided not to go work at that hippie commune.

He started reading...

Learning Montage

Leon reads about a technique called Return Oriented Programing (ROP)

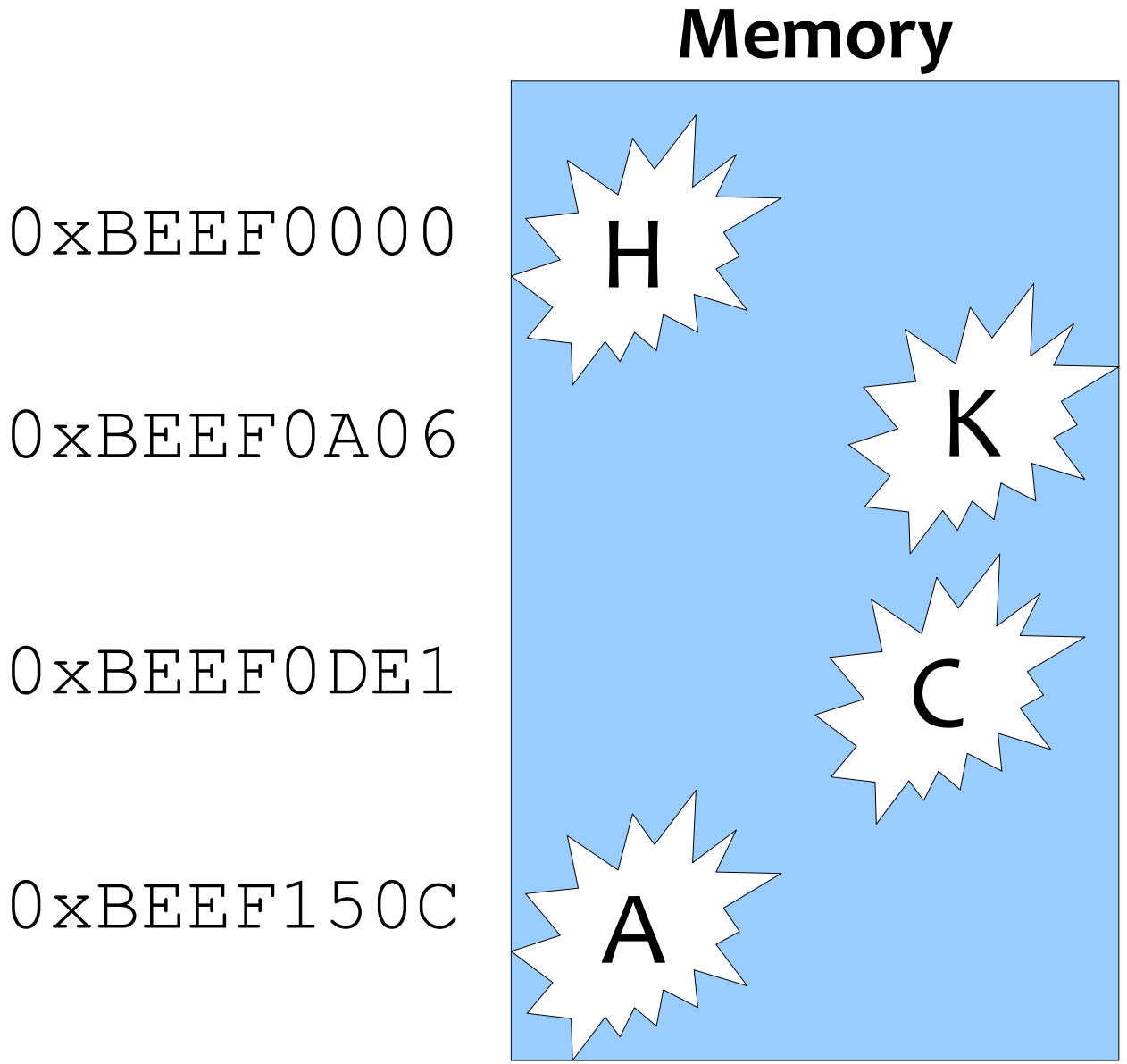
ROP reuses existing code to build a first-stage shellcode. By reusing existing code, it will all be located on executable memory pages.

Learning Montage

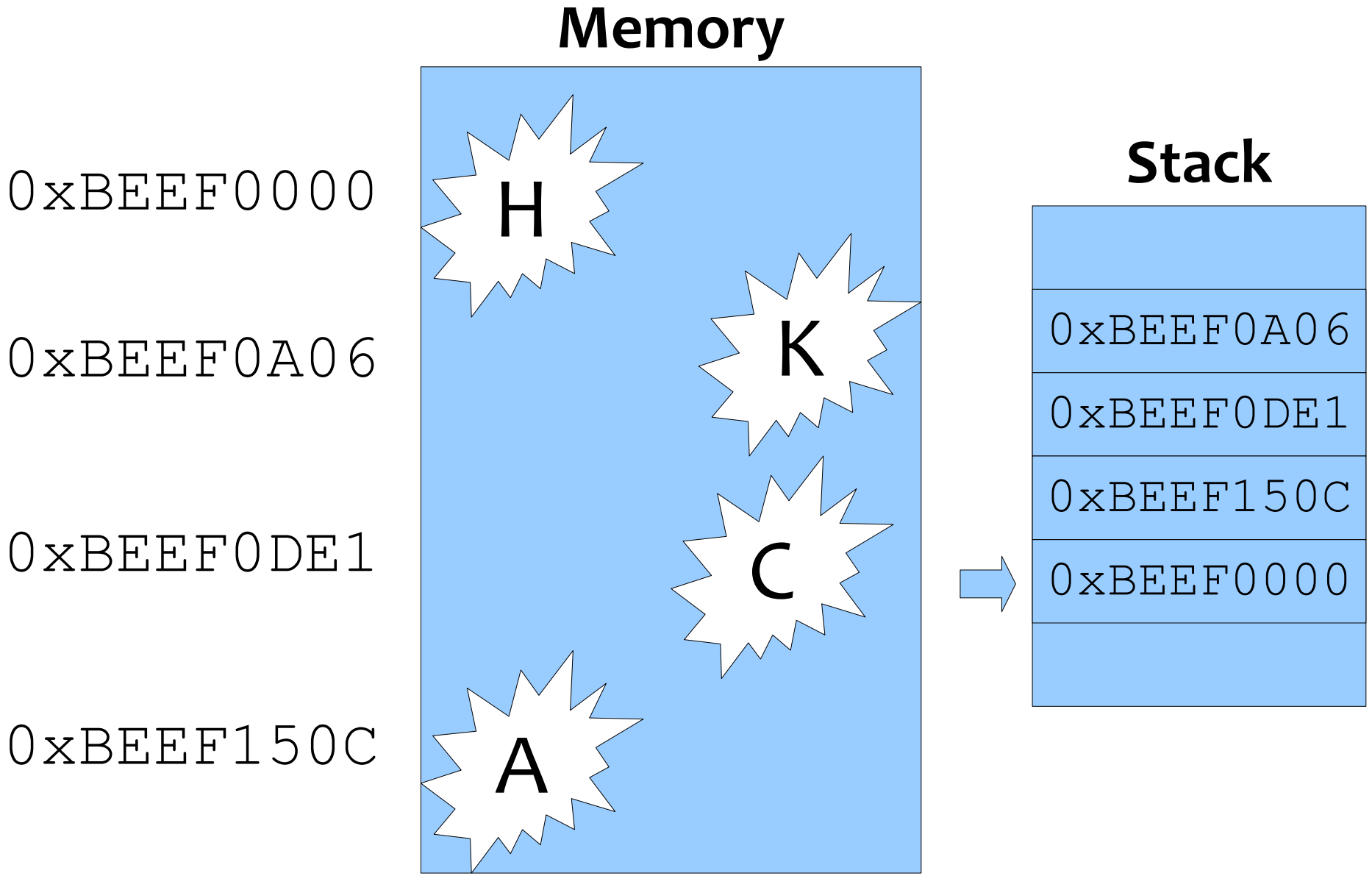
@drraid: I explained ROP to my gf, and she said "So it's like when you make a ransom note out of newspaper word cut-outs?" <-- brilliant analogy [1]

[1] Twitter is great. Follow me. @dionthegod

Aside: ROP


Memory
0xBEEF0000
0xBEEF0A06
0xBEEF0DE1
0xBEEF150C
H
K
C
A

Aside: ROP


Memory
0xBEEF0000
0xBEEF0A06
0xBEEF0DE1
0xBEEF150C
H
K
C
A
Stack
0xBEEF0A06
0xBEEF0DE1
0xBEEF150C
0xBEEF0000

Learning Montage

Leon thinks ROP is the cat's pajamas.

Unfortunately, with ASLR it is impossible (well...) to find those ~~letters~~ pieces of code.

Learning Montage

Leon is leet. He thinks a information leak that gave him an address could be used to figure out where the code was loaded.

Plan B

Step 1. Find an information leak.

Plan B

Step 2. ROP

Plan B

Step 3. Profit?

Raise? Bonus?

Leon is no longer thinking big.
Leon doesn't know how to find an info leak with his 589 fuzzers.

Leon Plans Ahead

Leon decides to have a 3rd cup of coffee and formulate another plan (Plan C).

You know... just in case he can't find that leak.

Plan C: The Birthing

Leon has read Alex Sotirov and Mark Dowd's BlackHat 2008 paper.

Leon thinks it would be cool to find a new way to spray the heap with executable pages (like Alex and Mark).

Plan C

Step 1. Find a way to heap spray on executable pages.

Plan C

Step 2. Exploit as before. DEP no longer blocks the original exploit – the pages are executable.

Plan C

Step 3. Profit?

Leon has no place to begin on with this idea, at first. So, he is less than optimistic.

Where to begin?

Leon has read that Adobe is still incorporating an SDL into their development process.

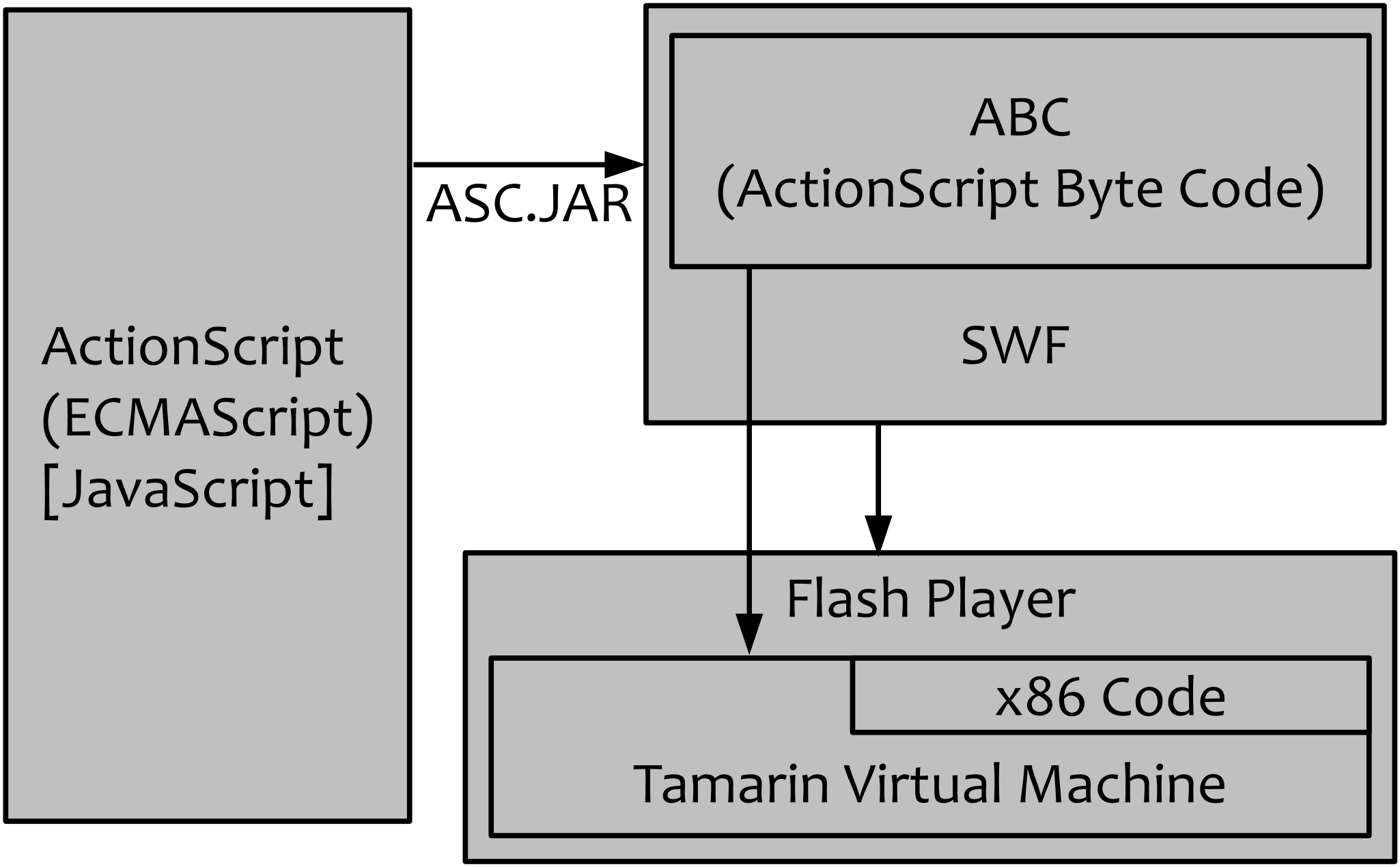
Current products are SDL-less. Leon gets that tingle again. Leon decides Adobe Flash Player is a fine place to start looking.

Adobe Flash (Savior of the Universe!)

Through a very cool blog[1], Leon learns that the ActionScript core interpreter for Adobe Flash player is open source. More tingle.

[1] http://dion.t-rexin.org/notes [2]
[2] Sometime before the end of the weekend, it will be moving to: http://blog.semantiscope.com

Aside: How Flash "Works"



Leon has a *Flash* of intelligence

Leon notices that Tamarin uses tagged pointers to store ActionScript objects within the virtual machine.

Aside: Tamarin Atoms

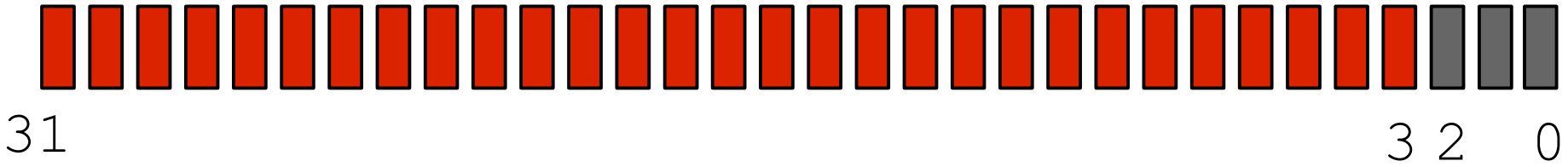
The ActionScript Bytecode encodes a dynamically typed language.

Objects within the VM have types associated at runtime. This means all runtime objects must store both value and type.

Aside: Tamarin Atoms

One well known technique is to carry some type information in the least significant bits.

Aside: Tamarin Atoms



Value

Type Tag

Untagged – 0
Object – 1
String – 2
Namespace – 3
“undefined” – 4
Boolean – 5
Integer – 6
Double – 7

Aside: Tamarin Atoms

Some objects store their actual value in the Atoms value bits (Integers, Booleans); others store a pointer there (Objects, Strings, Namespaces, Doubles).

Aside: Tamarin Atoms

The Integer `42` → Atom `0x00000156`

`42 = 0x2a = b00101010`

Integer tag `= 6 = 0x6 = b110`

Atom `= 00101010 110`

Atom `= 001 01010110 = 0x156`

Aside: Tamarin Atoms

Object obj is allocated at `0x0000ffe0`

obj → Atom `0x0000ffe1`

Object tag = `1` = `0x1` = `b001`

Atom = `0x0000ffe0 | b001`

Atom = `0x0000ffe1`

Leon is on a roll

While reading the Tamarin code, Leon notices a general purpose hashtable. This hashtable maps Atoms to Atoms. It is also possible to iterate over the table.

Just one more key fact...

The hashtable uses the value bits of the key Atom for the hash value.

This means an integer that is less than the size of the hashtable will be placed at that offset in the table.

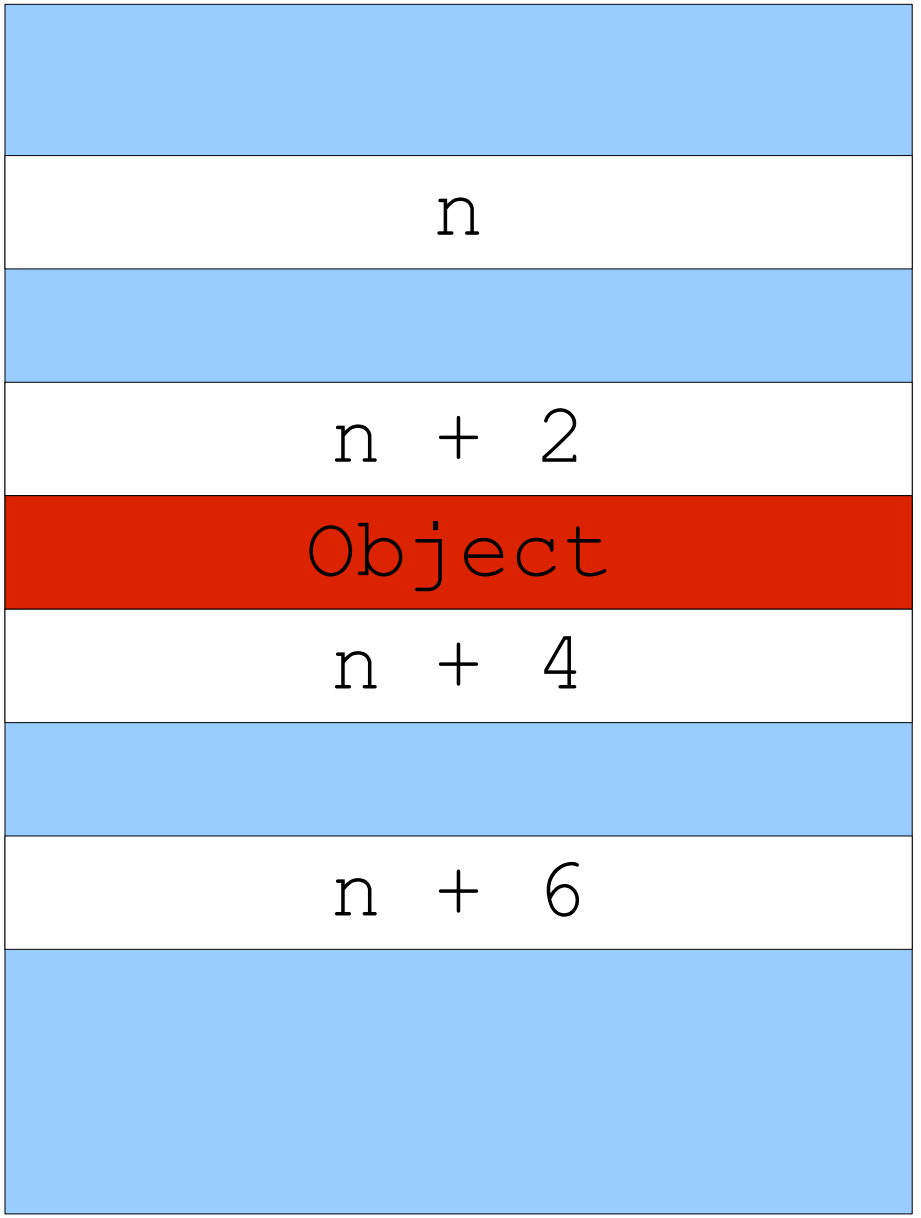
Leon makes the cognitive leap

The hashtable is ordered by Atom value.

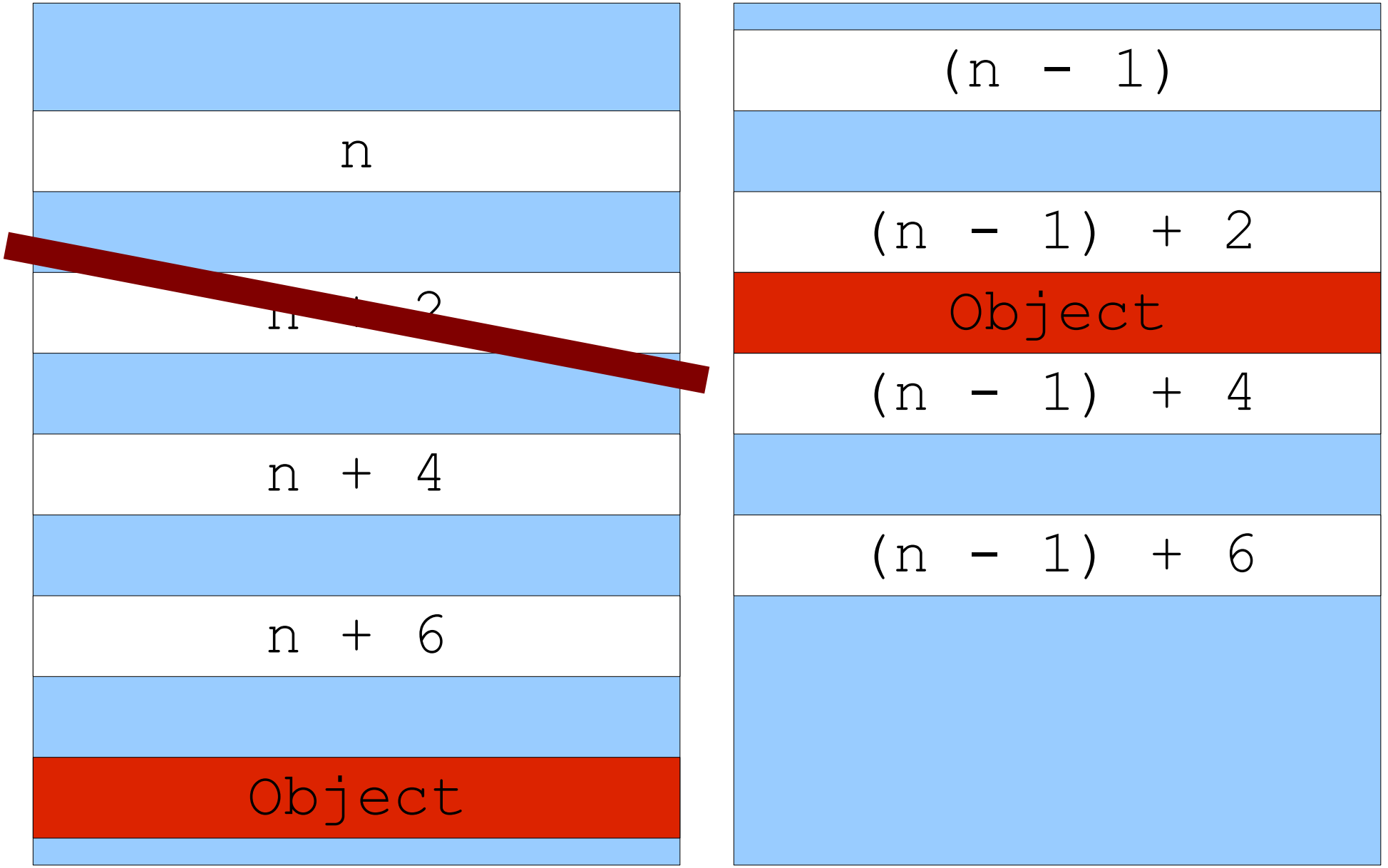
The Atom value of an Object is a memory address. If Atom types can be mixed in the hashtable, integers are compared to addresses.

The ordering of a mixed HT leaks address bits.

Aside: Pointer Inference



Aside: Pointer Inference



Aside: Pointer Inference???

All the details are in the paper. Sample code is (will be) available on the website:

http://www.semantiscope.com/research/BHDC2010

Any questions, shoot me an e-mail.

Leon is leakin'

Leon now has a way to leak many bits of an address.

Err… What address?

On the heap

The leaked address is “some random heap address.”

Depending on the type, there are some controllable fields, but this isn't the function pointer leak Leon had hoped for.

Leon sees the glass half full

The info leak is nice, but Leon isn't sure where to go with it. He has a few ideas, but decides to have a go at Plan C.

Despite the lack of good info leak, Leon is unflagging in his pursuit of code execution.

ROP with a typewriter?

While reading about ROP, Leon mused that it might be possible to use the JIT compiler engine to create code with useful bits of code to steal.

Aside: JIT

Just-in time (JIT) compilation is the process of converting an intermediate bytecode into native code at runtime.

Aside: JIT

For our purposes, we're talking about turning ActionScript bytecode into x86 machine code at runtime.

This means we are writing code that is later executed.

Aside: JIT

“Wait a minute, I thought code sections weren't writeable!”

Ah. I lied. The OS provides functions to mark memory as writable, readable, and executable.

Aside: JIT

The JIT engine allocates sections of memory and marks them as executable.

These sections are also marked writable for a short period of time while the JIT engine writes the machine code.

What was that about a typewriter?

Leon thought it might be possible to coerce the JIT engine to spit out code that contained useful code in the constants.

RET all up ins

ROP relies on RET (0xc3) values – Leon wanted to pass immediate values to the ActionScript that contained the 0xc3.

Leon makes a(nother) breakthrough

He starts by feeding a single constant integer to the ActionScript interpreter. By watching the JIT results, Leon hopes to find that it passes through unchanged.

Constant values!

Leon passes this to the ActionScript compiler:

```
var f = function () {
   var a = 0x00414141 ^
           0X00424242;
   return a;
};
f();
```

Constant values!

Leon sees this produced by the JIT compiler in the Tamarin VM:

```
mov eax, 0x00414141
xor eax, 0x00424242
```

Leon envisions profit

After some experimentation, Leon realizes that this code pattern can be extended arbitrarily – a long XOR expression in ActionScript will result in a very compact x86 instruction stream.

Leon gets a better idea.

A better idea

What if Leon could point EIP into the middle of this JITed code? What if he could encode all his shellcode into those immediates used in the XOR expression?

Leon *really* smells the profit.

Aside: JIT Crafting

Let's go back to the sample code:

```
var f = function () {
  var a = 0x00414141 ^
          0X00424242;
  return a;
};
f();
```

Aside: JIT Crafting

Remember, that code was turned into this:

```
03470069: mov eax, 0x00414141
0347006E: xor eax, 0x00424242
```

Or:

```
03470069:
B8 41 41 41 00 35 42 42 42 00
```

Aside: JIT Crafting

Imagine entering at `0x0347006A` instead of `0x03470069`.

Execution would be attacker controlled opcodes.

Now, we need to find a way to ensure execution continues along this attacker controlled stream.

Aside: JIT Crafting

The MOV and XOR instructions are single byte instructions. If the last byte of the attacker controlled immediates (aka MSB) were a semantic NOP instruction that required a single byte operand, those MOV and XOR instruction bytes would be skipped.

Aside: JIT Crafting

```
B8 D9D0543C     mov eax, 3C54D0D9
35 586AF43C     xor eax, 3CF46A58

B8 D9 D0 54 3C 35 58 6A F4 3C

D9D0            fnop
54              push esp
3C 35           cmp al, 35
58              pop eax
6A F4           push -0C
```

Tiny Shells

Leon now knows how to write ActionScript such that if he redirects EIP to the middle of the JIT output, he gains arbitrary execution.

He has to code 1-3 bytes at a time. Not impossible, just painful.

Stage-0?

Leon notices the JIT code will contain the Object pointer if used in the function and also contains function pointers in the Flash Player binary.

Using these pointers, Leon writes stage-0 shellcode (1-3 bytes at a time).

String → Execute

The stage-0 shellcode marks a region of memory executable using the function pointer to calculate the address of the VirtualProtect call in the Tamarin VM.

Then it copies the value of an ActionScript String into this region and jumps to it.

Address?

Leon is happy.

His next step is to find a way to fill memory with these JITed functions.

With a large spray, he should be able to predict an address that contains his Jishcode.

Aside: Jishcode

JITed code that contains that stage-0 shellcode.

Jishcode.

Leon looks at loaders

The ActionScript API contains calls to dynamically load SWF files from network resources **and** from ByteArray objects.

Leon's first idea is to create giant in memory SWF files with loads of identical functions.

Leon sprays instead

Leon comes to his senses and realizes it would be far easier to repeatedly load the same SWF with a single function.

As long as a reference to that function is maintained, the JITed code will stay in memory.

Leon quickly integrates

Leon quickly integrates this exploit code with the Flash Gordon (Yes, the movie from 1980) based game he designed from Plan A.

Leon is the picture of joy

It works!

Leon rejoices. Leon celebrates.

Dion demos.

Leon shows his boss

Leon's boss wants to justify the time spent developing the hashtable leak.

“Leon, can you use the leak to reliably gain the address of one of the JITed regions?”

When Leon's boss asks a question, it means he wants a real answer.

Leon gets cozy with the Tamarin Heap

Leon spends sometime reading Tamarin code.

He puts instrumentation points into Flash Player to make sure he understands.

Leon gets that look in his eye...

Aside: Tamarin Heap

The Tamarin VM implements its own heap on top of the OS VM calls.

The heap is similar to the Windows heap.

Aside: Tamarin Heap

Smallish requests are tried in the appropriate bucket of like sized pieces.

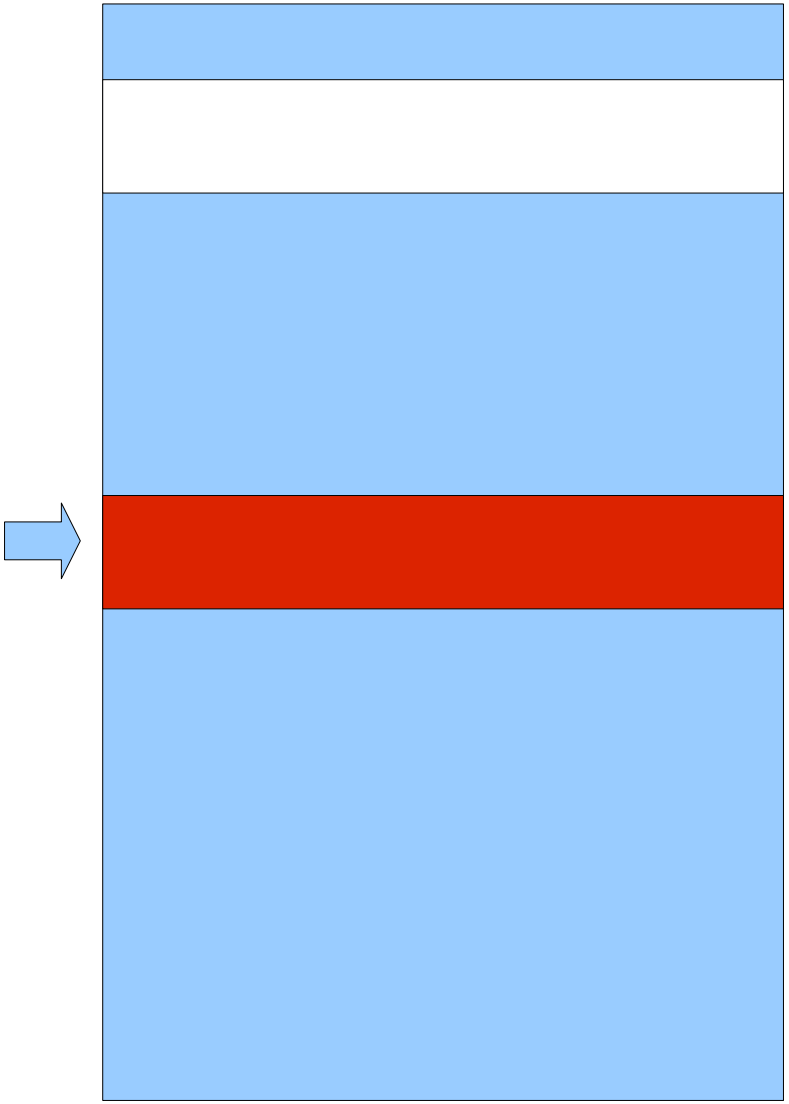
If a large request is needed, or the bucket style allocator cannot make one available, the heap expands.

Aside: Tamarin Heap

The heap attempts to expand contiguously and tries to allocate `0x01000000` bytes at a time.

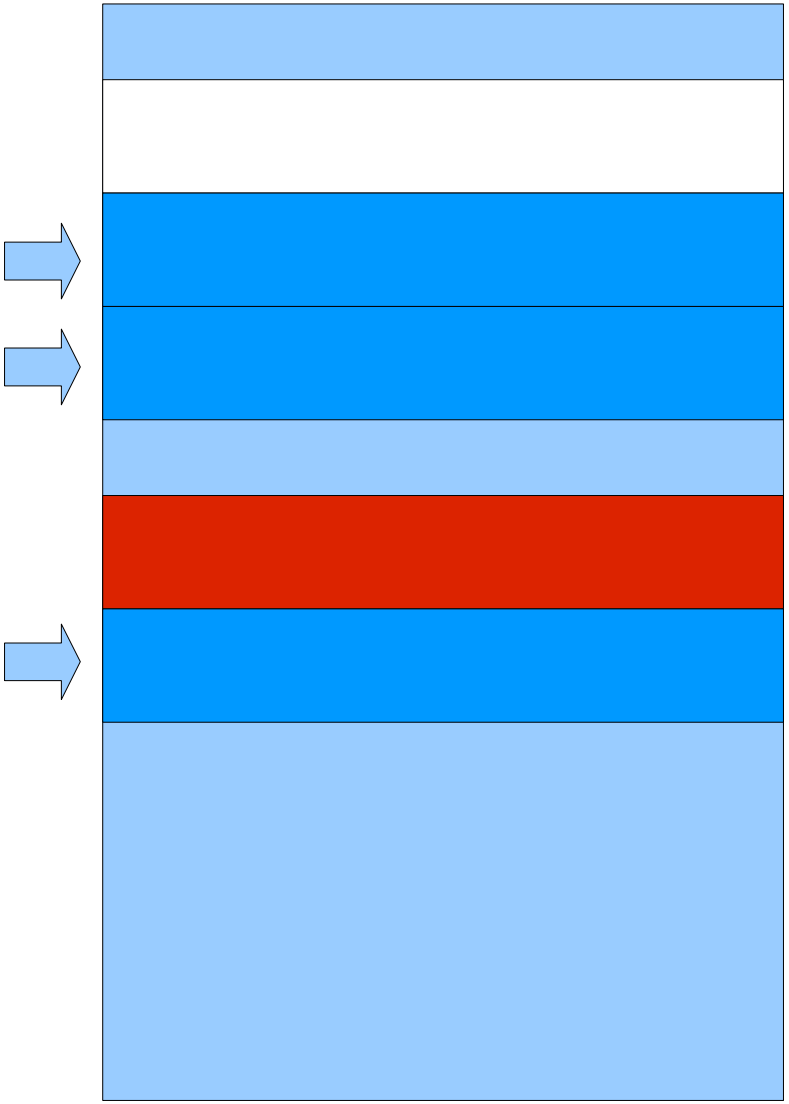
Leon's Steps to Success: 1

Open an SWF with enough bytecode to force an allocation of `0x01010000`



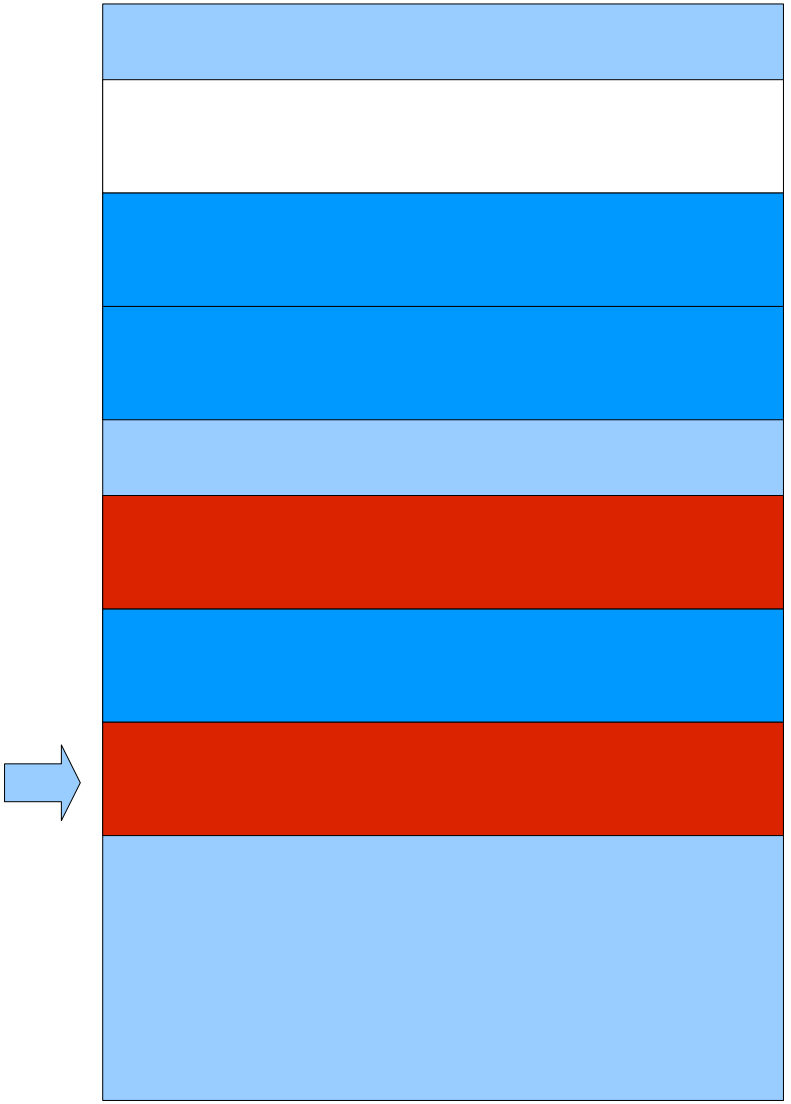
Leon's Steps to Success: 2

Allocate many small ActionScript Objects. Force the heap to grow past the big block.



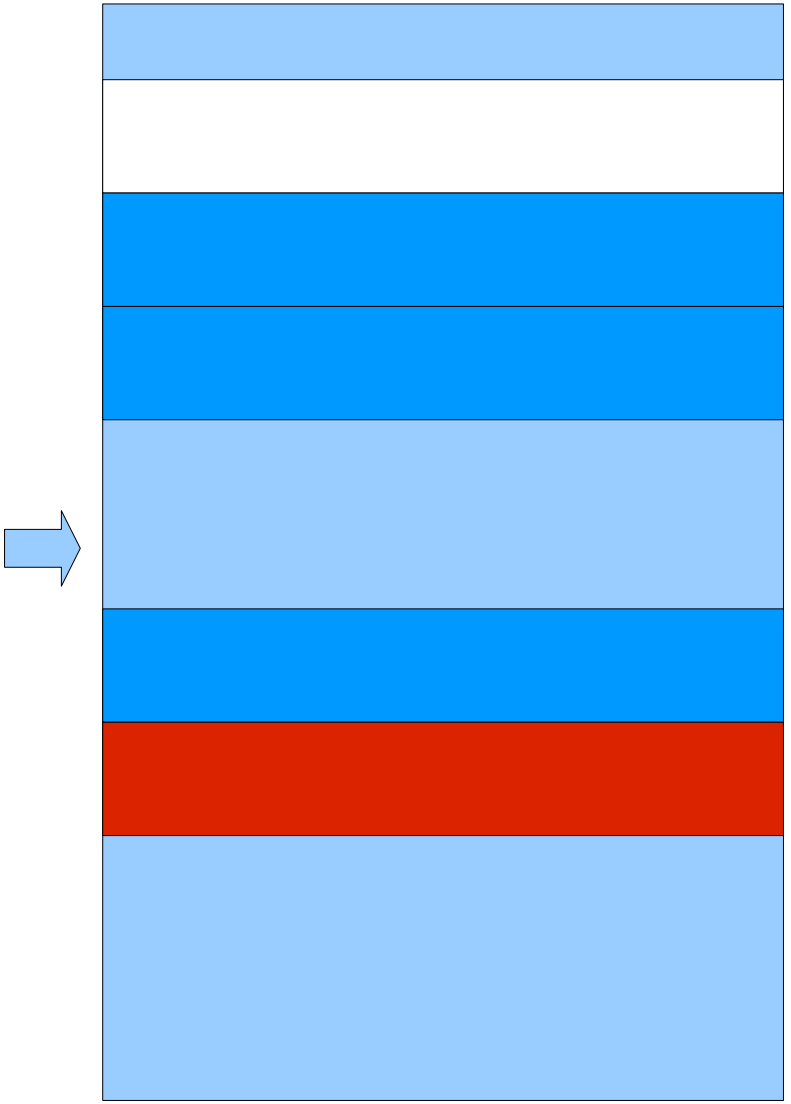
Leon's Steps to Success: 3

Open an SWF with enough bytecode to force an allocation of `0x00FF0000`



Leon's Steps to Success: 4

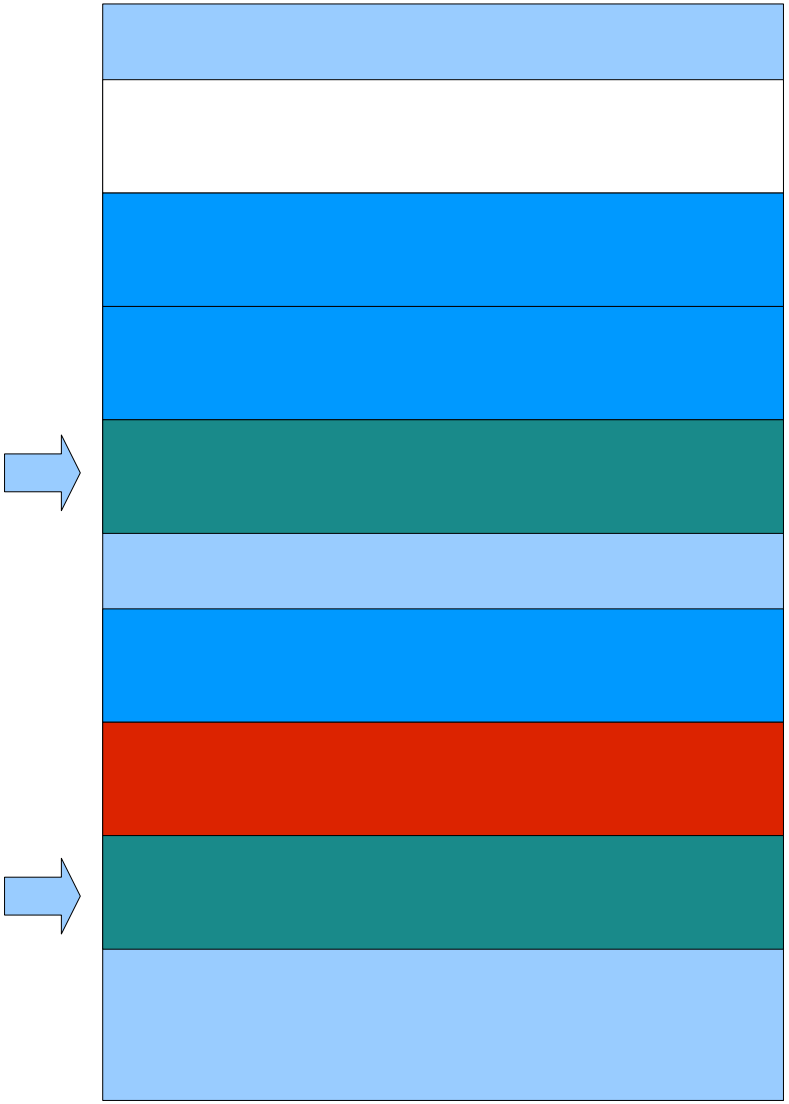
Release that first SWF.



Leon's Steps to Success: 5

Spray more than 0x01000000 of small ActionScript Objects.

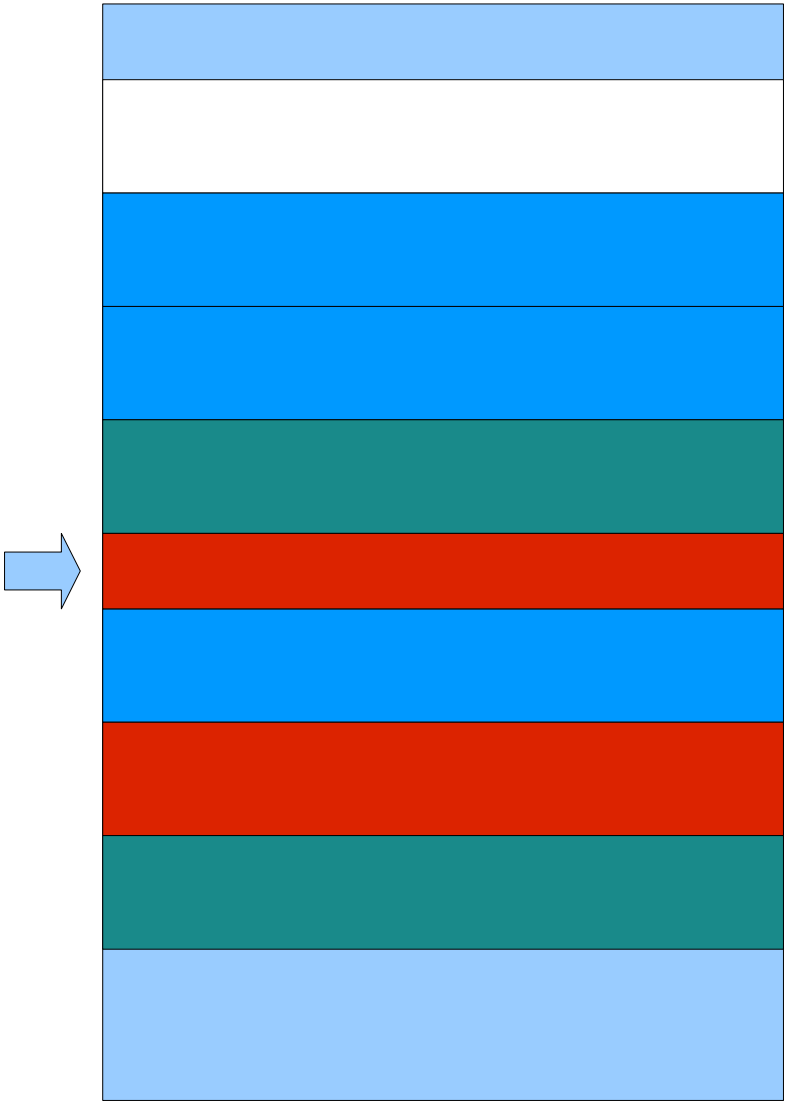
Maintain these in a linked list style structure.



Leon's Steps to Success: 6

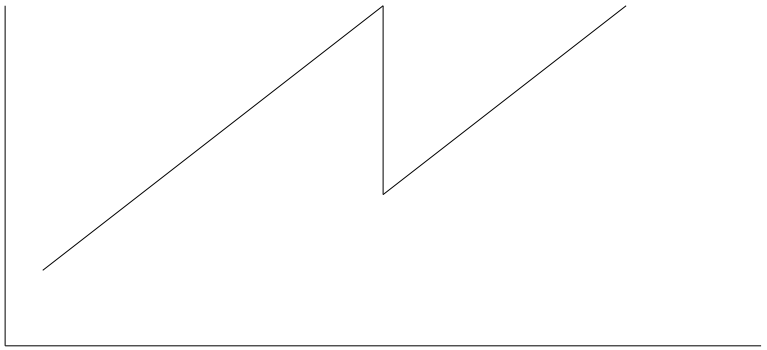
Perform a JIT spray.

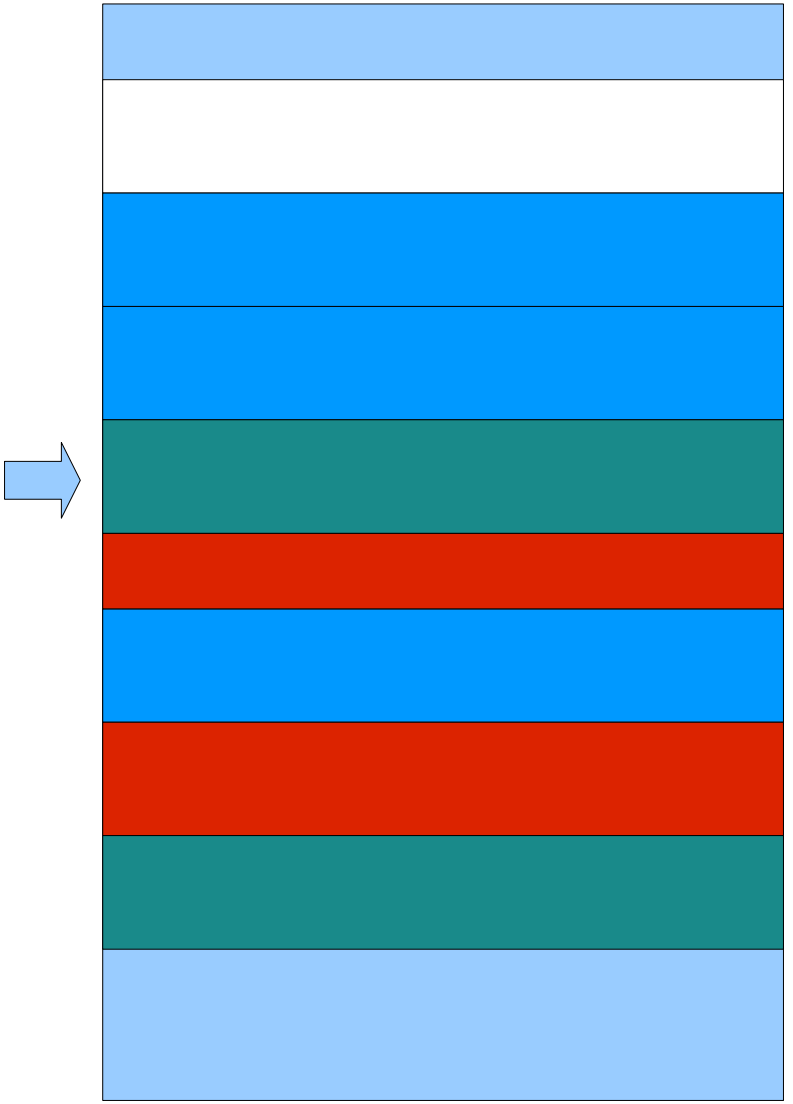
Each script will be small and reserve the minimum (0x00010000 bytes)



Leon's Steps to Success: 7

Iterate over that linked list from step 5. Performing the pointer inference.





Leon decides “Nah”

Leon figures 8 minutes is too long to wait – his Flash Gordon game isn't THAT good.

He tells his boss no.

His boss doesn't care.

Leon get's that promotion. Leon wins.

Mitigation mitigation mitigations

How do we prevent JIT Spraying?

Disable JIT.
Make the output non-deterministic.

Mitigation mitigation mitigations

How do we prevent this weird leak?

Use an XOR cookie to scramble the reference atoms in the hastable? (Thanks, antijon)

Make the iterator go over all keys of the current type before iterating over keys of the next type? (Thanks, Eric)