OSY.SSI [2018] [12]

In the news...

- TODO

In the last episode

- Our program died, killed by a ruthless `SEGFAULT` in a dark alley
- An unnamed culprit, a maniac, frolics in our streets, free to murder again
- We must investigate, shed light on this crime and put and end to this folly
- Or is there more to it that meets the eye?

In the last episode

- Our program died, killed by a ruthless `SEGFAULT` in a dark alley
- An unnamed culprit, a maniac, frolics in our streets, free to murder again
- We must investigate, shed light on this crime and put and end to this folly
- Or is there more to it that meets the eye?

Ok so basically we are going to repeat the murder again and again, watching closely what happens with `gdb`.

Three questions about `eip`

Three questions about `eip`

1. Why is `eip` suddenly `0x42424242`?

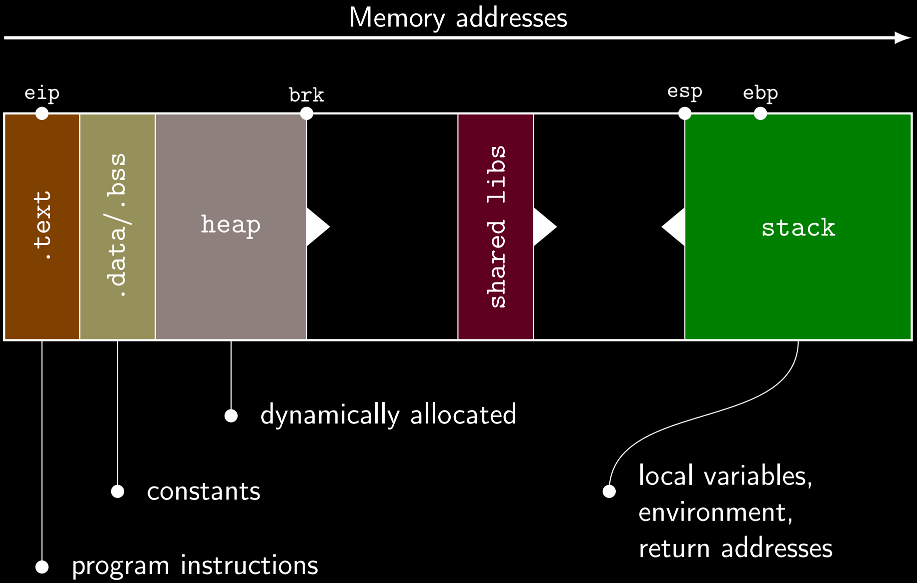
Three questions about `eip`

1. Why is `eip` suddenly `0x42424242`?
2. Can we control its value more precisely?

Three questions about `eip`

1. Why is `eip` suddenly `0x42424242`?
2. Can we control its value more precisely?
3. What can we do with it?

Let's take some distance



1: The Reason

- When a program calls a function, it stores the current `eip`...

1: The Reason

- When a program calls a function, it stores the current `eip`... on the stack.

1: The Reason

- When a program calls a function, it stores the current `eip`... on the stack.
- When we overwrite the stack, we end up overwriting this saved `eip`
- After the function terminates, the program wants to return to where it was, but...
- All it can see now is BBBBBBB...

2: The Force

- First, find the minimum number of `B` so that `eip = 0x42424242` on crash.

2: The Force

- First, find the minimum number of `B` so that `eip = 0x42424242` on crash.
- Now, make it so that `eip = 0x41424344`. (this corresponds to `ABCD`).

2: The Force

- First, find the minimum number of `B` so that `eip = 0x42424242` on crash.
- Now, make it so that `eip = 0x41424344`. (this corresponds to `ABCD`).

At this point, we can completely control `eip`.

2: The Force

- First, find the minimum number of `B` so that `eip = 0x42424242` on crash.
- Now, make it so that `eip = 0x41424344`. (this corresponds to `ABCD`).

At this point, we can completely control `eip`.

That means we can send the program to run any code we want.

2: The Force

- First, find the minimum number of `B` so that `eip = 0x42424242` on crash.
- Now, make it so that `eip = 0x41424344`. (this corresponds to `ABCD`).

At this point, we can completely control `eip`.

That means we can send the program to run any code we want.

What code do we want?

Table of Contents

All praise Helix

(this is a reference to *Twitch plays Pokemon*)

Task: Make the program run the `hidden_gem` function!

All praise Helix

(this is a reference to *Twitch plays Pokemon*)

Task: Make the program run the `hidden_gem` function!

This is control flow hijack. We can basically run any code from this program or one of its libraries. Especially interesting if the program has priviledged access.

Chaining

Are we limited to one hijack? Or can we get back the control, and send the program somewhere else?

Chaining

Are we limited to one hijack? Or can we get back the control, and send the program somewhere else?

Yes we can!

Chaining

Are we limited to one hijack? Or can we get back the control, and send the program somewhere else?

Yes we can!

Okay so force the program to Hail Helix several times. What happens if you run it 4 times?

Return-oriented programming

More generally, we call call any function with or without arguments, and chain with another call etc. until we're satisfied.

In fact, we don't need to call *functions*, anywhere in the code will do. But to get back the control after the first hijack, we need a `ret` or a `ret ret`.

A remarkable fact is that such instructions are very common, and large enough programs have plenty. We call them *gadgets*.

(For large enough programs, with high probability, gadgets are Turing-complete. In practice that means we can usually always do what we want)

Shellcodes

Ok so ROP works but is tricky to pull off.

Can we instead just write the attack code, and run it?

Shellcodes

Ok so ROP works but is tricky to pull off.

Can we instead just write the attack code, and run it?

Yes we can!

But to do that well we need two things: an attack code, and some more stack stuff.

1. Put the shell and the code in the shellcode

We write a small program launching a shell.

```
31 c9                             xor   %ecx,%ecx    ; ecx = 0
f7 e1                             mul   %ecx         ; eax = ecx*eax
51                                push  %ecx         ; 0
68 2f 2f 73 68                    push  $0x68732f2f ; //sh
68 2f 62 69 6e                    push  $0x6e69622f ; /bin
89 e3                             mov   %esp,%ebx    ; ebx = esp
b0 0b                             mov   $0xb,%al     ; eax = 11
cd 80                             int   $0x80        ; syscall
```

This program calls `syscall(11, ['/bin', '/sh'], 0)`. The 11th syscall is `execv`.

So essentially this program runs a shell. Can we test it?

1. Put the shell and the code in the shellcode

We write a small program launching a shell.

```
31 c9                        xor   %ecx,%ecx    ; ecx = 0
f7 e1                        mul   %ecx         ; eax = ecx*eax
51                           push  %ecx         ; 0
68 2f 2f 73 68               push  $0x68732f2f  ; //sh
68 2f 62 69 6e               push  $0x6e69622f  ; /bin
89 e3                        mov   %esp,%ebx    ; ebx = esp
b0 0b                        mov   $0xb,%al     ; eax = 11
cd 80                        int   $0x80        ; syscall
```

This program calls `syscall(11, ['/bin', '/sh'], 0)`. The 11th syscall is `execv`.

So essentially this program runs a shell. Can we test it? (Yes we can!)

2. Some more stack stuff

Ok so now we have a program to run, but how do we run it *in the victim program*?

2. Some more stack stuff

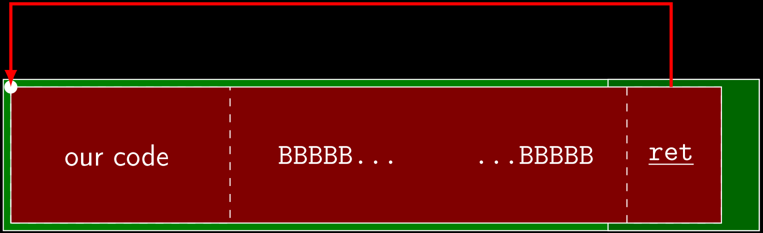
Ok so now we have a program to run, but how do we run it *in the victim program*?

1. We can send `eip` anywhere in memory.
2. Everything is somewhere in memory.
3. What if instead of `BBBBB...` we used our code?

2. Some more stack stuff

Ok so now we have a program to run, but how do we run it *in the victim program*?

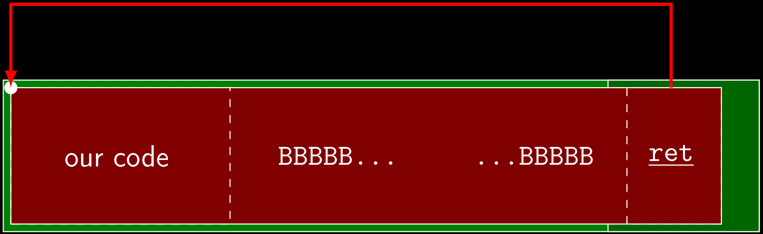
1. We can send `eip` anywhere in memory.
2. Everything is somewhere in memory.
3. What if instead of `BBBBB...` we used our code?



2. Some more stack stuff

Ok so now we have a program to run, but how do we run it *in the victim program*?

1. We can send `eip` anywhere in memory.
2. Everything is somewhere in memory.
3. What if instead of `BBBBB...` we used our code?



Demo! Demo! Demo!

A few things to adjust

- If you test the script as is, it only works with gdb (because env)
  (I have a script that accounts for this automatically)
- If you try to attack the SUID version, ASLR will kick in
  (We can cheat and disable it `sudo sysctl -w kernel.randomize_va_space=0`)
- If you try to attack the SUID version, you may not be root
  (Most shells ignore setuid)
- This was only the 32 bit version
  (64 bit is no different, just a bit longer)

This is just a matter of adjusting, none of which is out of reach.

Different ways to smash the stack

A lot of information is on the stack, not only local variables, but also e.g. environment variables

Environment variables are not restricted and modify stack addresses

Demo

Shellcoding and Oulipo

- Designing good shellcodes is harder than it looks
- Many constraints, and easily thwarted (unlike ROP)
- But maaaaaany examples available / goto `https://www.exploit-db.com/`

Table of Contents

Putting things together

Ok so let's summarise what we know:

- The ability to inject data may enable us to run code
  - For instance, run a shell (typically, a connect-back shell) or a RAT
- This is in practice often the case (several such vulnerabilities found every day)
- There are other ways to control a system (e.g. standard bugs)

Putting things together

- In particular, software that reads files or network information is exposed

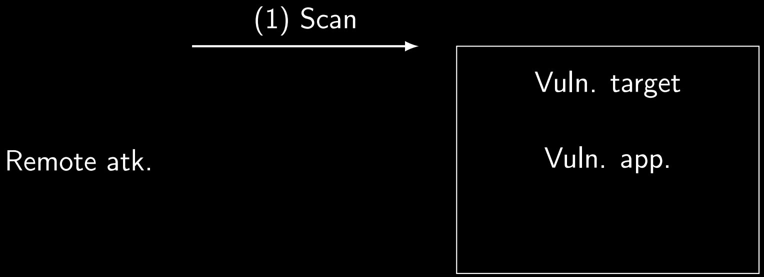
Putting things together

- In particular, software that reads files or network information is exposed
  - Think PDF and PDF readers, Flash and Flash animations
  - Think network devices, think Apache servers
- Sometimes these vulnerabilities are patched. Sometimes the patches aren't applied.

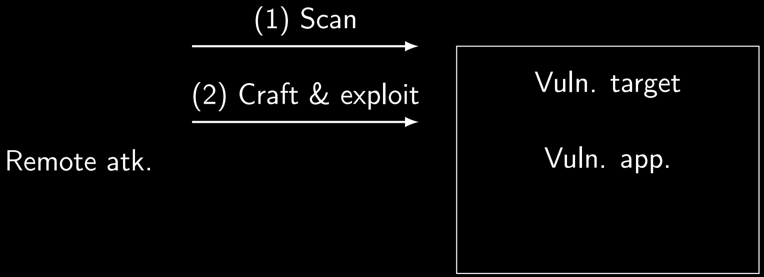
Putting things together

- In particular, software that reads files or network information is exposed
    - Think PDF and PDF readers, Flash and Flash animations
    - Think network devices, think Apache servers
- Sometimes these vulnerabilities are patched. Sometimes the patches aren't applied.
    - Remember `nmap` can help you find software versions
- We know how to send arbitrary network messages over the Internet

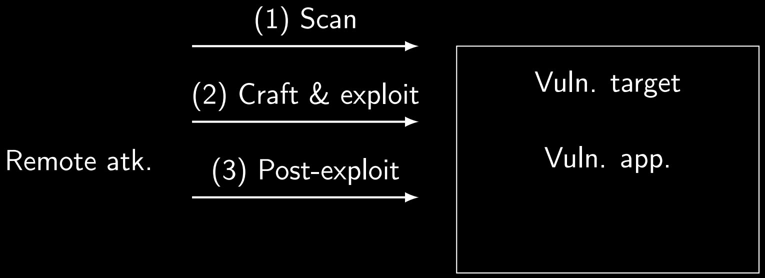
Exploitation overview



Exploitation overview



Exploitation overview



Exploitation overview

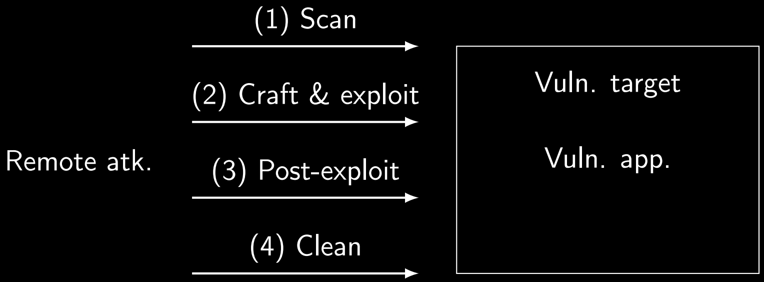


Table of Contents

Smashing the stack in 2018

EVERYBODY KNOWS ABOUT STACK OVERFLOWS

- More or less efficient detection techniques (canaries...)
- Non-executable stack (NX, W$\hat{X}$)
- Memory randomisation (ASLR)
- Signed code
- ASCII-armoured addresses
- “Memory-safe” languages

Does this mean we cannot play anymore :(?

Conditions for exploitation

We need

- Attacker-controlled data
- Which is interpreted, or misinterpreted, as code or commands (escaped, overflowed, race condition, etc.)
- With enough leeway to do something interesting
- Using as little extra knowledge as possible

Any candidate?

JIT: Java, Flash, `asm.js`, etc.

To gain performance, some systems leverage just-in-time compilation (JIT), which is essentially writing code and then executing it.

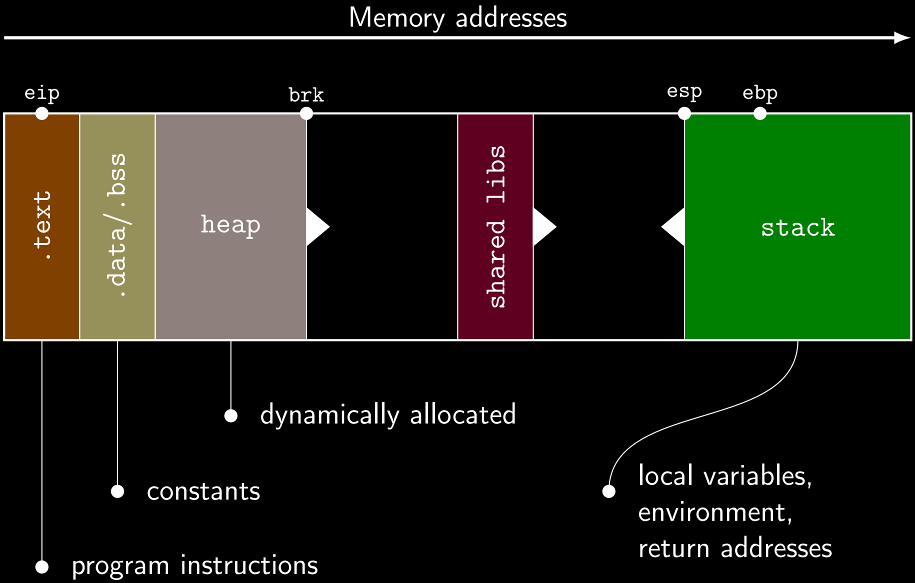
JIT: Java, Flash, `asm.js`, etc.

To gain performance, some systems leverage just-in-time compilation (JIT), which is essentially writing code and then executing it.

These guys make life easier for us. Sometimes, they enjoy exceptions from the restrictive rules hampering other programs.

It's not a mystery that attackers target them preferentially.

Refresher: memory structure



Can't hack the stack? Reap the heap!

Heap-based overflow is still young and fresh!

Can't hack the stack? Reap the heap!

Heap-based overflow is still young and fresh!

The heap stores dynamically allocated arrays, so it's dynamic and moving and a bit harder to get right.

Demo

Can't hack the stack? Reap the heap!

Heap-based overflow is still young and fresh!

The heap stores dynamically allocated arrays, so it's dynamic and moving and a bit harder to get right.

Demo

The heap is massively used to instantiate “objects” in “object-oriented” programming.

Refresher: Object-Oriented Programming

- Old notion from the 1950s, inspired by Aristotle: code segments are “objects” which
    - Are instances of an abstract “class” from which they get a common structure
    - Are endowed with properties (attributes) and dispositions (methods)

- To make this work, a lot of adjustments are needed
    - Scope (attributes and methods may be more or less hidden from other instances or classes)
    - Complex inheritance patterns (e.g. Square/Rectangle), Liskov principle
    - Dependency injection
    - State encapsulation, constructors, destructors, move & copy

Refresher: Object-Oriented Programming

"Object-oriented programming is an exceptionally bad idea which could only have originated in California."

– Edsger Dijsktra (1989)

"Java is the most distressing thing to happen to computing since MS-DOS."

– Alan Kay (1997)

"Object-oriented programming offers a sustainable way to write spaghetti code."

– Paul Graham (2003)

"(...) don't screw things up with any idiotic "object model" crap."

– Linus Torvalds (2007)

"Object-oriented programming (...) is a powerful idea (...) not always the best idea."

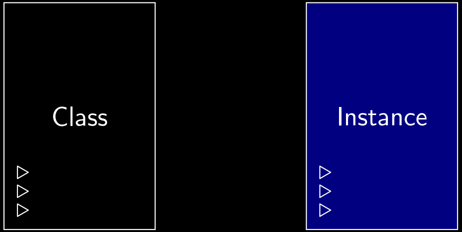
– Rob Pike (2012)

"Object-oriented programming is a terrible technique because it encourages two very damaging practices: taxonomic classification and encapsulation of state"

– Some guy on the Internet (2014)

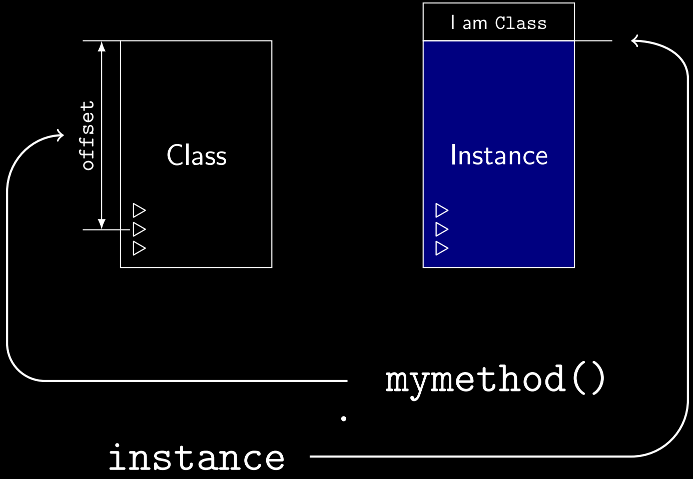
One typical scenario

- Describe a “class”
- Instanciate an object of this class
- Call a method from that instance
- Call a method from that instance
- ...
- Call a method from that instance
- Destroy the instance and go on with your life



```
instance.mymethod()
```

Technical aside: dynamic dispatch, vtable



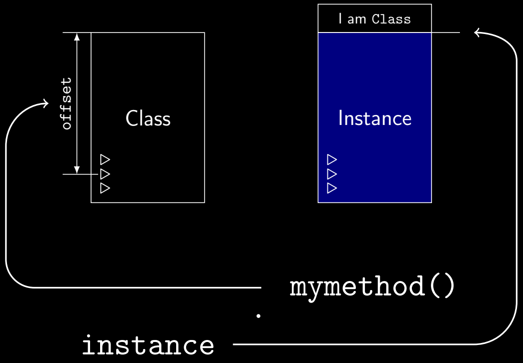
Use-after-free

Sometimes a program creates an object and deletes it (e.g., exception handling code), but some part of the program is not aware of this deletion.

So the other part of the program may call this object. In general, this does not crash, because the code *is still there, in memory, untouched*.

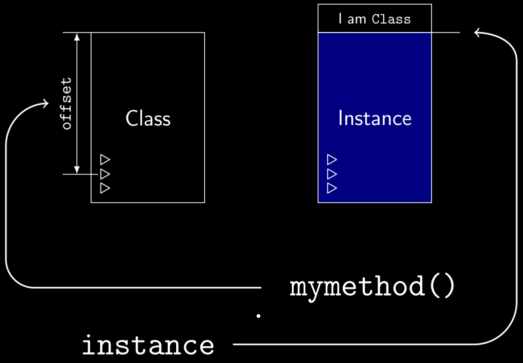
Note that the heap *must* be executable otherwise OOPs don't work.

Heap exploitation through vtables, basic idea



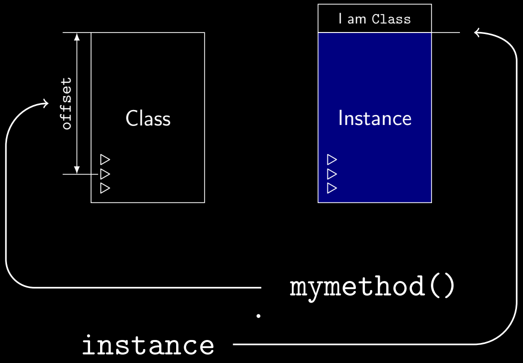
- Instanciate an object
- Call a method
- Call a method
- ...

Heap exploitation through vtables, basic idea



- Instanciate an object
- Call a method
- Call a method
- ...
- Cause the object to be destroyed
- Call a method

Heap exploitation through vtables, basic idea



- Instanciate an object
- Call a method
- Call a method
- ...
- Cause the object to be destroyed
- Call a method

Can we rewrite the object's memory to replace it with our own code?

On subtelty

Sometimes, we have to be subtle and tactful and precise and delicate.

On subtelty

Sometimes, we have to be subtle and tactful and precise and delicate.

This is not one of those times.

On subtelty

Sometimes, we have to be subtle and tactful and precise and delicate.

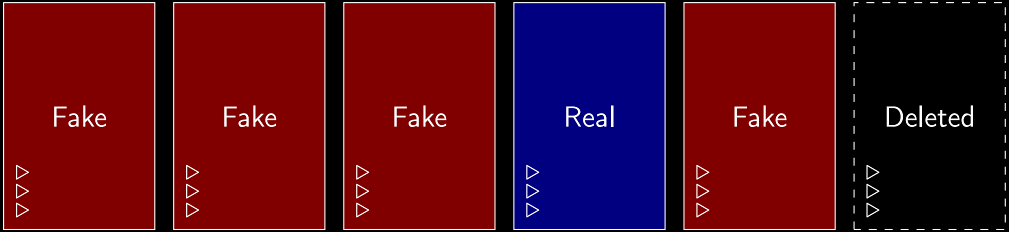
This is not one of those times.

Introducing: heap spreaying!

1309

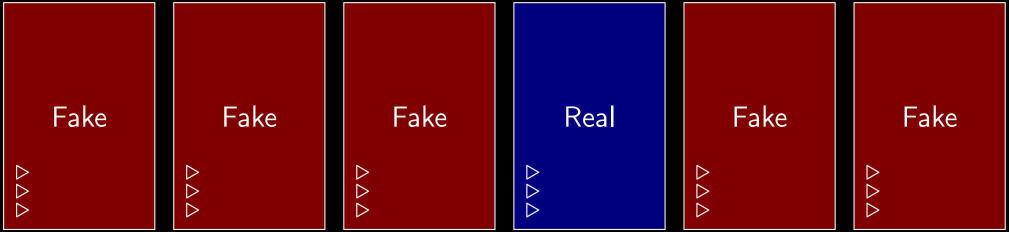
Heap spraying: how it works

Use the target to allocate on the heap



Heap spraying: how it works

Use the target to allocate on the heap



Spraying and UAF exploitation

1. Cause the target to create the object
2. Cause the target to free/delete the object
3. Spray the heap like your life depends on it! This hopefully replaces the deleted object's code with your code.
4. Cause the target to use-after-free; it will then run your code.

Note: not subtle, so target mammoths who consume loads of memory anyway (Internet browser, PDF readers, video players, etc.)

Note: Can a priori be triggered from any program, through any means of spraying. For instance, a javascript code running in an ad in a different browser.

Demo

Exercise: control flow hijack

```
struct toystr {
 void (* message)(char *);
 char buffer[20];
};


coolguy = malloc(sizeof(struct toystr));
lameguy = malloc(sizeof(struct toystr));
coolguy->message = &print_cool;
lameguy->message = &print_meh;

printf("Input coolguy's name: ");
fgets(coolguy->buffer, 300, stdin);
coolguy->buffer[strcspn(coolguy->buffer, "\n")] = 0;

printf("Input lameguy's name: ");
fgets(lameguy->buffer, 20, stdin);
lameguy->buffer[strcspn(lameguy->buffer, "\n")] = 0;

coolguy->message(coolguy->buffer);
lameguy->message(lameguy->buffer);
```

Summary

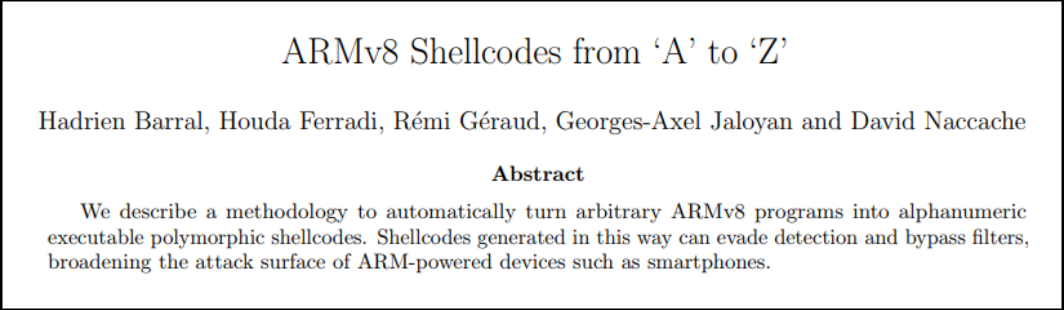
- Control memory, control everything
- Do not trust metaphors or mental models
- Consequences of integrity fail can be dramatic, even small overflows can lead to ACE
- Mitigations are hard and slow, and hardware-specific!
- Possibility of running arbitrary code with super-user privileges from across the Internet using a fake origin address

Shellcoding as an art form

Some very perverted people put a lot of effort to write exploits for various architectures that evade detections techniques and pass as legitimate inputs.

Shellcoding as an art form

Some very perverted people put a lot of effort to write exploits for various architectures that evade detections techniques and pass as legitimate inputs.



ARMv8 Shellcodes from 'A' to 'Z'

Hadrien Barral, Houda Ferradi, Rémi Géraud, Georges-Axel Jaloyan and David Naccache

Abstract

We describe a methodology to automatically turn arbitrary ARMv8 programs into alphanumeric executable polymorphic shellcodes. Shellcodes generated in this way can evade detection and bypass filters, broadening the attack surface of ARM-powered devices such as smartphones.

More!

- Type confusion
- Integer under and overflow
- Floating point issues
- Race conditions (Unix maze technique)
- The swiss cross

Okay that's enough