Hadrien Barral
Georges-Axel Jaloyan

3COLE NORMALE SUPÉRIEURE,
PSL RESEARCH UNIVERSITY

D3FC0N

3MOJI 5HELLCODING: 🛠️, 🧌, AND 🤯

WHO WE ARE

- Hadrien Barral: Hacker  . I like hacking stuff.
- Georges-Axel Jaloyan: PhD ~~student~~. I reverse open source binaries.
- This is our third/second talk at D3FC0N.



WHAT THIS IS ABOUT

- Some generic 🛠️ and methods
- ... with lots of emoji 🧌
- ... to write 🤯 shellcodes.

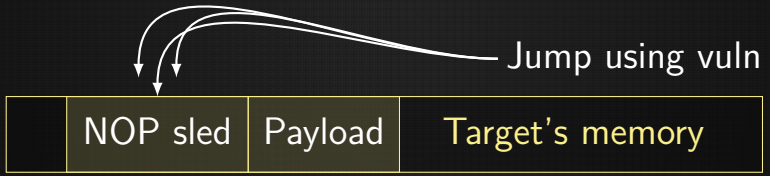
1. WHY WOULD ANYONE WRITE CONSTRAINED SHELLCODES?

SHELLCODE 101

For those hiding in the back

- Code that you wrote (or found) in the target's memory
- That gives you some power (e.g. pop a shell)
- That you can jump to using some vuln (e.g. buffer overflow, UAF...)

Usual scenario: send a carefully crafted string to the target and profit.



SHELLCODE 102

Not as convenient as you may think

- If treated as a C string, no \x00
- If treated as input, no whitespace
- Other constraints (regex)
- Breaks easily (string escaping)
- Does not really look legit 🤔



HAVING FUN WITH X86

rix, Phrack 57, 2001 + Basu et al., ICISS, 2014

Constrained shellcoding on x86 is easy:

ASCII	IA32
'P'	push %eax
'h'	push imm32
'X'	pop %eax
'a'	popa
'A'	inc %eax
'H'	dec %eax
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'8'	cmp r/m8 r8
'0'	xor r/m8 r8

HAVING FUN WITH X86

rix, Phrack 57, 2001 + Basu et al., ICISS, 2014

Constrained shellcoding on x86 is easy:

- `push`/`pop`/`inc`/`dec` have single letter instructions!

ASCII	IA32
'P'	push %eax
'h'	push imm32
'X'	pop %eax
'a'	popa
'A'	inc %eax
'H'	dec %eax
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'8'	cmp r/m8 r8
'0'	xor r/m8 r8

HAVING FUN WITH X86

rix, Phrack 57, 2001 + Basu et al., ICISS, 2014

Constrained shellcoding on x86 is easy:

- `push`/`pop`/`inc`/`dec` have single letter instructions!
- `jmp` and `cmp` are available

ASCII	IA32
'P'	push %eax
'h'	push imm32
'X'	pop %eax
'a'	popa
'A'	inc %eax
'H'	dec %eax
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'8'	cmp r/m8 r8
'0'	xor r/m8 r8

HAVING FUN WITH X86

rix, Phrack 57, 2001 + Basu et al., ICISS, 2014

Constrained shellcoding on x86 is easy:

- `push`/`pop`/`inc`/`dec` have single letter instructions!
- `jmp` and `cmp` are available
- `xor` with many operands are available

ASCII	IA32
'P'	push %eax
'h'	push imm32
'X'	pop %eax
'a'	popa
'A'	inc %eax
'H'	dec %eax
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'8'	cmp r/m8 r8
'0'	xor r/m8 r8

HAVING FUN WITH X86

rix, Phrack 57, 2001 + Basu et al., ICISS, 2014

Constrained shellcoding on x86 is easy:

- `push`/`pop`/`inc`/`dec` have single letter instructions!
- `jmp` and `cmp` are available
- `xor` with many operands are available
- 🤦 for x86-64: just prefix `0x48` (= `H`) to every instruction

ASCII	IA32
'P'	push %eax
'h'	push imm32
'X'	pop %eax
'a'	popa
'A'	inc %eax
'H'	dec %eax
't'	je short disp8
'u'	jnz short disp8
'v'	jbe short disp8
'8'	cmp r/m8 r8
'0'	xor r/m8 r8



CRAZY PEOPLE EVEN MADE ENGLISH SHELLCODES

Mason et al., CCS, 2009

demo demo demo

ALPHANUMERIC SHELLCODING: TAKING RISCS

RISC – *reduced* instruction set computer...

- No single character instructions anymore
- Few addressing modes – in particular no memory to memory
- Heavy constraints on operands

Previous technique does not work on RISC architectures! 😱

ALPHANUMERIC SHELLCODING: TAKING RISCS

RISC – *reduced* instruction set computer...

- No single character instructions anymore
- Few addressing modes – in particular no memory to memory
- Heavy constraints on operands

Previous technique does not work on RISC architectures! 😱

Compilation / Emulation / Unpacking

THE ``COMPILATION'' WAY

Idea:

- Compile assembly code to a constrained instruction set

Pros:

- Feasible for one-instruction set computers (e.g Movfuscator on x86)

Cons:

THE ``COMPILATION'' WAY

Idea:

- Compile assembly code to a constrained instruction set

Pros:

- Feasible for one-instruction set computers (e.g Movfuscator on x86)

Cons:

- Does not work when the constraints are mainly on the operands and not on the opcodes
- ...nobody wants to devote their life to writing such a compiler 😰

THE ``EMULATION'' WAY

Younan et al., Phrack 66, 2009

Idea:

- Write an interpreter for some language

Pros:

- Reusable for different payloads
- e.g. Younan's ARMv7 alphanumeric Brainfuck interpreter

Cons:

THE ``EMULATION'' WAY

Younan et al., Phrack 66, 2009

Idea:

- Write an interpreter for some language

Pros:

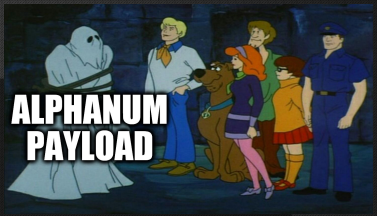
- Reusable for different payloads
- e.g. Younan's ARMv7 alphanumeric Brainfuck interpreter

Cons:

- Harmfulness now relies on the interpreter...

THE ``UNPACKING'' WAY

Barral et al., ISPEC 2016



Idea:

- Encode payload in a constraint-compliant way (e.g. alphanumerically)

THE ``UNPACKING'' WAY

Barral et al., ISPEC 2016

Idea:

- Encode payload in a constraint-compliant way (e.g. alphanumerically)
- Identify high-level constraint-compliant constructs (zeroing/increasing registers, ...)



THE ``UNPACKING'' WAY

Barral et al., ISPEC 2016

Idea:

- Encode payload in a constraint-compliant way (e.g. alphanumerically)
- Identify high-level constraint-compliant constructs (zeroing/increasing registers, ...)
- Use this to write a minimal unpacker that decodes and executes payload ☠️





ARMV8 DEMO

demo demo demo

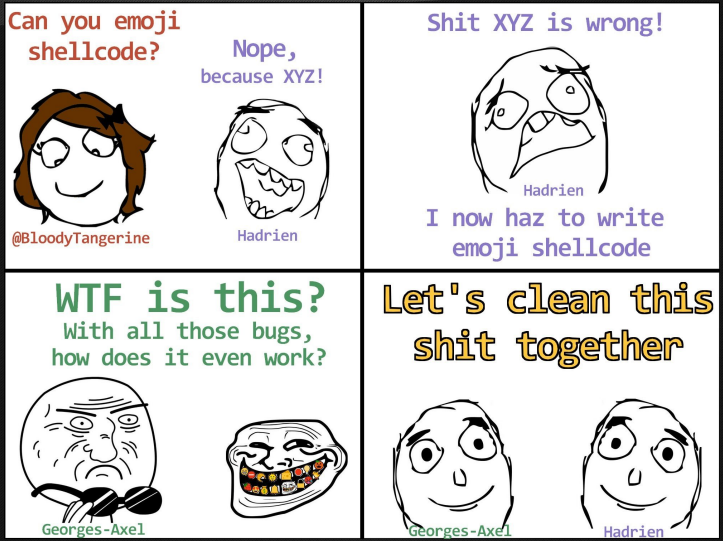
ALPHANUMERIC IS ``SOLVED''

🎉🎉🎉 Even works for RISC-V! 🎉🎉🎉

The ABC of Next-Gen Shellcoding, DEF CON 27

(protip, it is our previous talk)

ONCE UPON A TIME... (TRUE STORY)







2. INSPECTOR GADGET 





WHAT IS AN EMOJI?



WHAT IS AN EMOJI?





(or more likely a mess 💩 ...)

WHAT IS AN EMOJI?

✨ Unicode (1987) to the rescue ✨

> "Unicode is a standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems."
>
> — Wikipedia

WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A,

WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, 主,

WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A,  ,  ,

WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, 🈷, 🧍,  ,

WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, ま, 🎋, 🂪, 😃,

WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, ま, 🕯, 🂫, 😀, 𓀀,



WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, ま, 🧍, 🃏, 😃, 🏴🏴🏴🏴🏴🏴🏴, 🔮,





WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

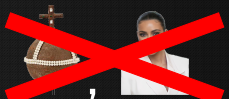
A,  ,  ,  ,  ,  ,  , 

WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, ま, 🕯, 🃏, 😃, 🏺, 🏈, 🤵

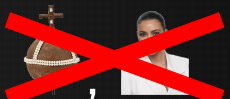


WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, ま, 🧍, 🤹, 😃, 🏺, 🏈, 🙋

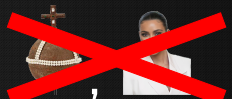


WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

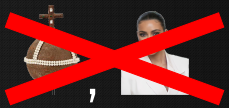
A, ま, 🧍, 🎭, 😃, 🏢, 🏈, 🎤



WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, 🕴, 🧍, 🃏, 😀,  , 

Definition:

If Unicode says it is a *qualified Emoji*, then it **is** an Emoji!

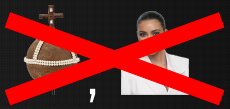
WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, ま, 🕯, 🎭, 😃, 🏢, 🏈, 🙋





Definition:

If Unicode says it is a *qualified Emoji*, then it **is** an Emoji!

In UTF-8, emojis at least 3 bytes 🙂,

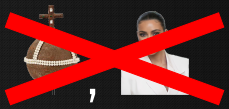
WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, ま, 🕯, 🎭, 😃, 🏢, 🏈, 🙋





Definition:

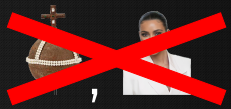
If Unicode says it is a *qualified Emoji*, then it **is** an Emoji!

In UTF-8, emojis at least 3 bytes 🙂, at most 35 👩‍🏾‍🤝‍🧑‍🏾.

WHAT IS AN EMOJI?

RTFM: https://unicode.org/reports/tr51/

In Unicode, you'll find:

A, 🕴, 🕯, 🃏, 😀,  , 

Definition:

If Unicode says it is a *qualified Emoji*, then it **is** an Emoji!

In UTF-8, emojis at least 3 bytes 🙂, at most 35 👩‍❤️‍👩.

And they add new emojis every year! Currently at Unicode version 14.

EXCUSE ME, DO YOU HAVE 2 MINUTES TO TALK ABOUT



EXCUSE ME, DO YOU HAVE 2 MINUTES TO TALK ABOUT



Features:

- Architecture of the future
- Simple, clean RISC Instruction Set Architecture
- Open Source ISA & Open Hardware*
- 2 and 4 byte instructions*, little-endian

*:Conditions may apply

EXCUSE ME, DO YOU HAVE 2 MINUTES TO TALK ABOUT



Features:

- Architecture of the ~~future~~ present
- Simple, clean RISC Instruction Set Architecture
- Open Source ISA & Open Hardware*
- 2 and 4 byte instructions*, little-endian

*:Conditions may apply

EXCUSE ME, DO YOU HAVE 2 MINUTES TO TALK ABOUT



Features:

- Architecture of the ~~future~~ present
- Simple, clean RISC Instruction Set Architecture
- Open Source ISA & Open Hardware*
- **2** and **4** byte instructions*, little-endian

*:Conditions may apply

HOW CAN I EXECUTE MY EMOJI?

HOW CAN I EXECUTE MY EMOJI?

- ✅ Alphanumeric x86: 'A' : `inc %eax`
  0x41

HOW CAN I EXECUTE MY EMOJI?

- ✅ Alphanumeric x86: ‘A’ : `inc %eax`
  `0x41`
- ✅ Alphanumeric ARM&RISCV: ‘4A0s’ : `csrc mip,sp`
  `0x34413073`

HOW CAN I EXECUTE MY EMOJI?

- ✅ Alphanumeric x86: ‘A’ : `inc %eax`
  `0x41`
- ✅ Alphanumeric ARM&RISCV: ‘4A0s’ : `csrc mip,sp`
  `0x34413073`
- ❓ Emoji RISCV:

HOW CAN I EXECUTE MY EMOJI?

- ✅ Alphanumeric x86: ‘A’ : `inc %eax`
  `0x41`
- ✅ Alphanumeric ARM&RISCV: ‘4A0s’ : `csrc mip,sp`
  `0x34413073`
- ❌ Emoji RISCV: ❓ : only 10 instructions
  `0xE29D93`

HOW CAN I EXECUTE MY EMOJI?

- ✅ Alphanumeric x86: 'A' : `inc %eax`
  `0x41`
- ✅ Alphanumeric ARM&RISCV: '4A0s' : `csrc mip,sp`
  `0x34413073`
- ❌ Emoji RISCV: ❓❓: still very few instructions
  `0xE29D93E29D93`

HOW CAN I EXECUTE MY EMOJI?

- ✅ Alphanumeric x86: 'A' : `inc %eax`
  `0x41`
- ✅ Alphanumeric ARM&RISCV: '4A0s' : `csrc mip,sp`
  `0x34413073`
- ❌ Emoji RISCV: ❓❓❓... : does not get better 😭
  `0xE29D93E29D93E29D93...`

HOW CAN I EXECUTE MY EMOJI?

- ✅ Alphanumeric x86: ‘A’ : `inc %eax`
  `0x41`
- ✅ Alphanumeric ARM&RISCV: ‘4A0s’ : `csrc mip,sp`
  `0x34413073`
- ❌ Emoji RISCV: ❓❓❓... : does not get better 😭
  `0xE29D93E29D93E29D93...`

We are stuck!

THE INTUITION...

```
auipc ra,0x979ff
```

THE INTUITION...

97 F0 9F 97

auipc ra,0x979ff

THE INTUITION...

97 F0 9F 97



auipc ra,0x979ff

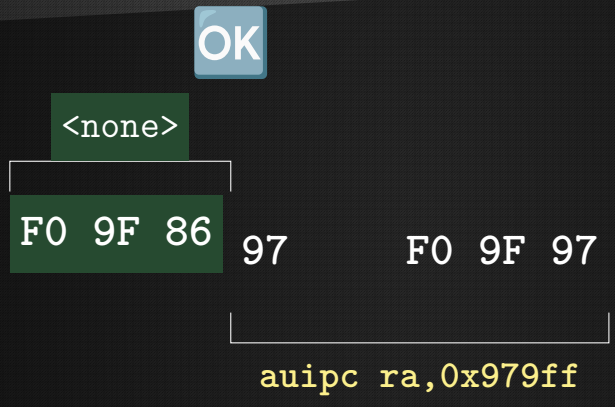
THE INTUITION...



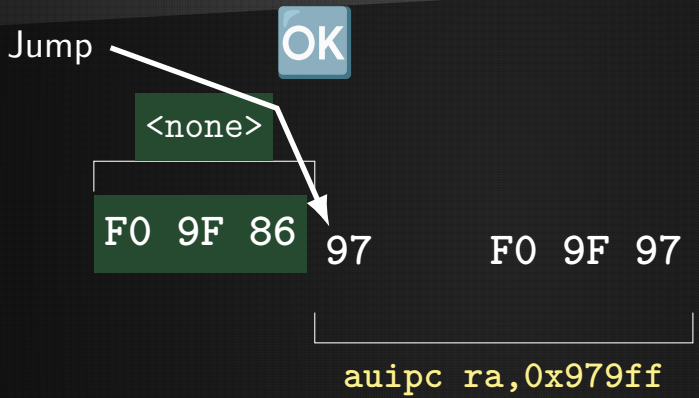
F0 9F 86 97 F0 9F 97

auipc ra,0x979ff

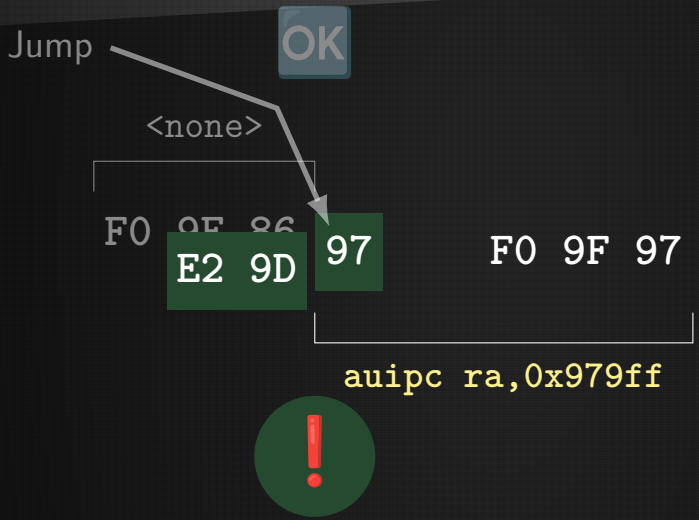
THE INTUITION...



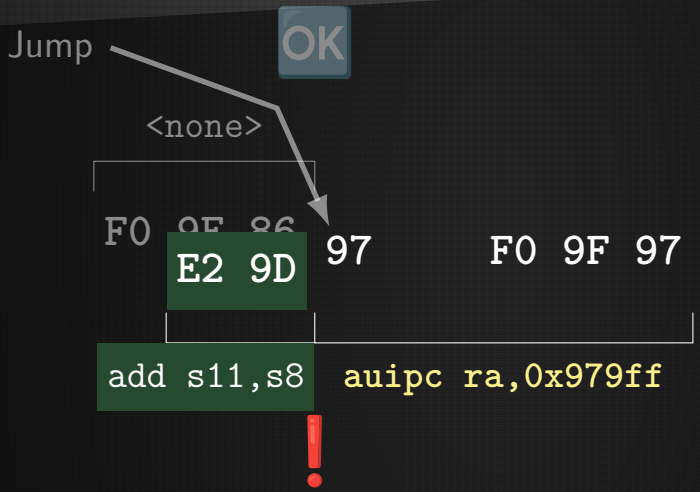
THE INTUITION...



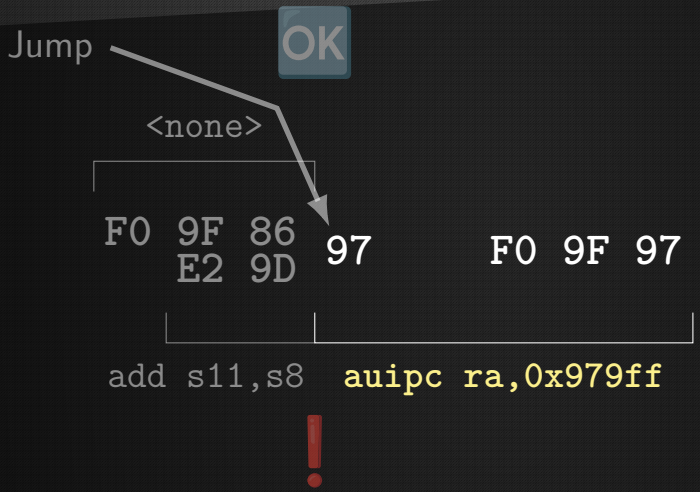
THE INTUITION...



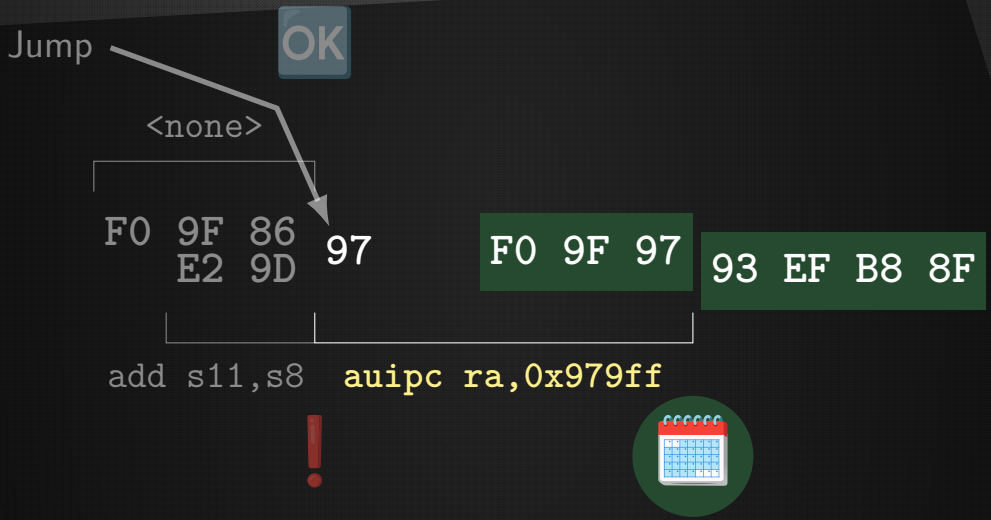
THE INTUITION...



THE INTUITION...



THE INTUITION...

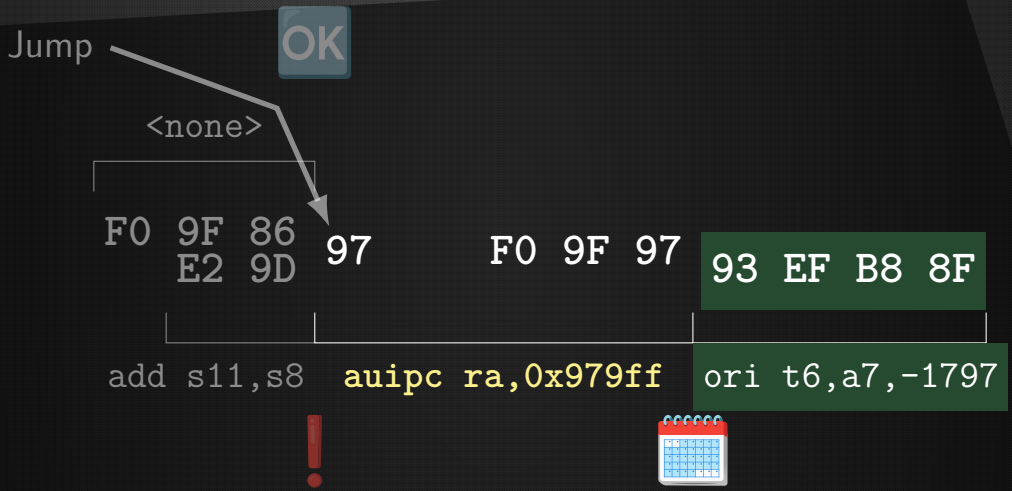


THE INTUITION...

Jump

OK

<none>

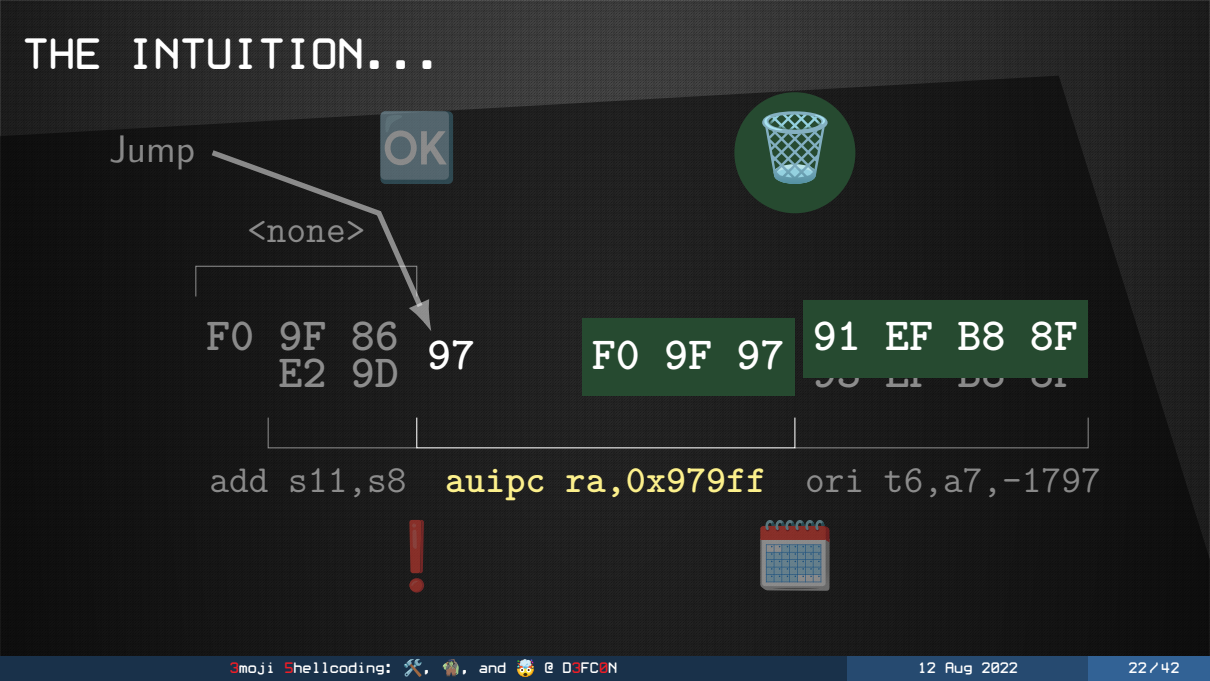


F0 9F 86 E2 9D 97 F0 9F 97 93 EF B8 8F

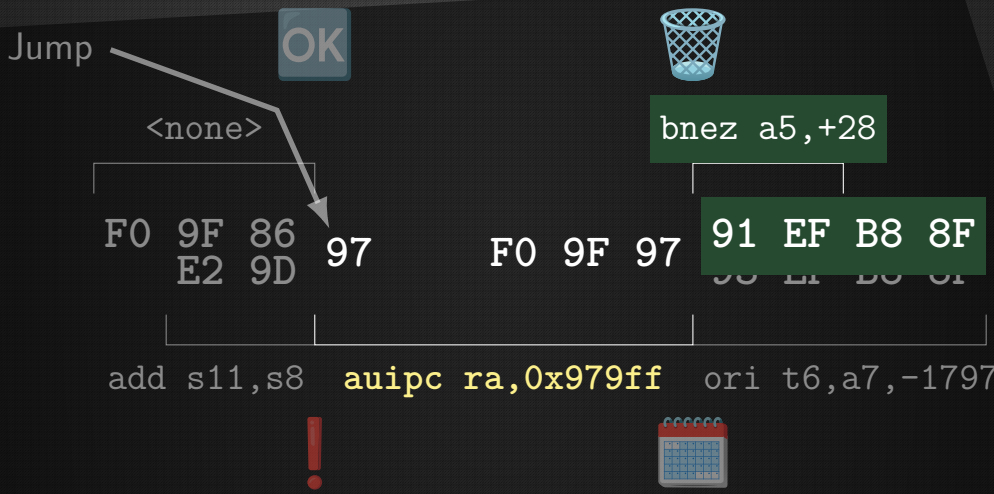
add s11,s8 auipc ra,0x979ff ori t6,a7,-1797

3moji Shellcoding: 🛠️, 🥷, and 🤯 @ D3FC0N 12 Aug 2022 22/42

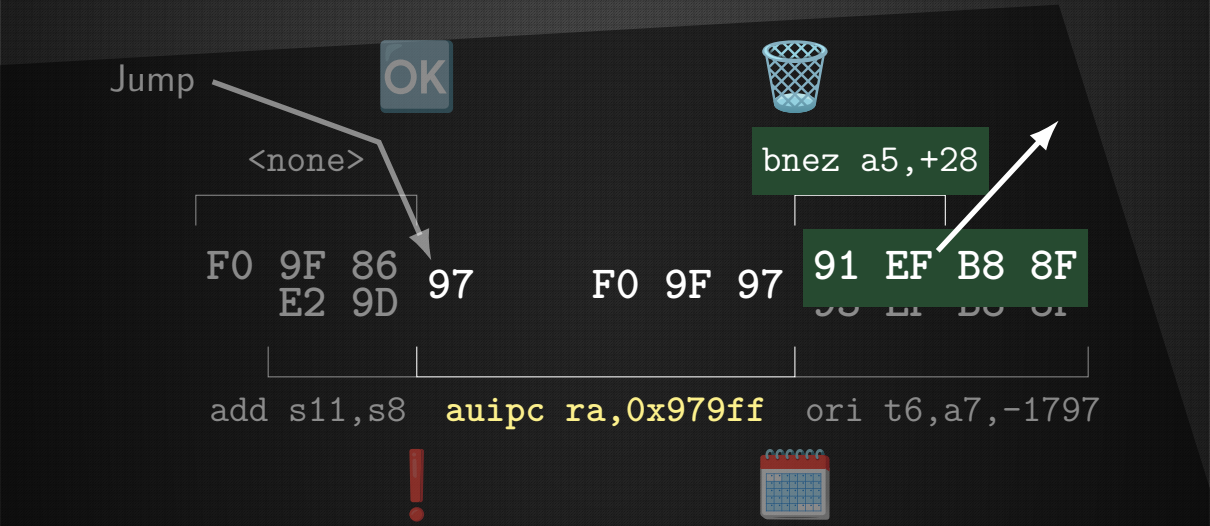
THE INTUITION...



THE INTUITION...



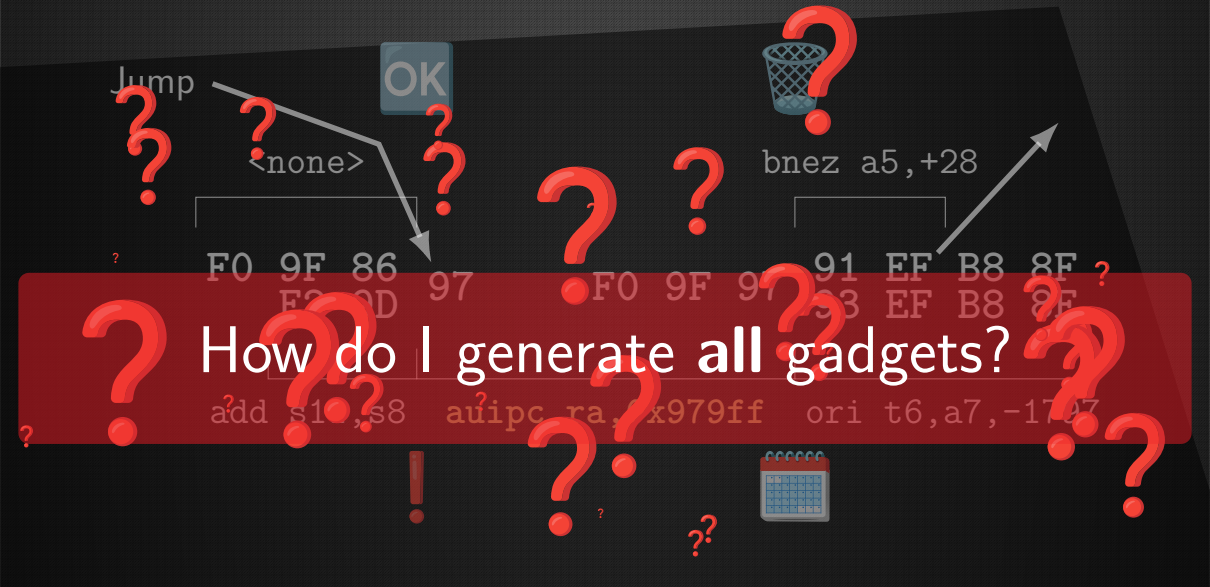
THE INTUITION...



THE INTUITION...



THE INTUITION...



How do I generate **all** gadgets?

EMOJI SHELLCODING IS A CODE-REUSE PROBLEM

Source: Trust me, bro.

- Return-Oriented Programming (Shacham et al., 2007):
  1. Scan target binary for reusable code snippets *(gadgets)*
  2. Build your shellcode with gadgets
  3. ???
  4. Profit

EMOJI SHELLCODING IS A CODE-REUSE PROBLEM

Source: Trust me, bro.

- Return-Oriented Programming (Shacham et al., 2007):
  1. Scan target binary for reusable code snippets *(gadgets)*
  2. Build your shellcode with gadgets
  3. ???
  4. Profit
- JIT spraying (Blazakis, 2010): create your own gadgets by controlling JITted bytecode.

```
var y = (
  0x3c54d0d9 ^
  0x3c909058 ^
  0x3c59f46a ^
  ...
```

is turned into:

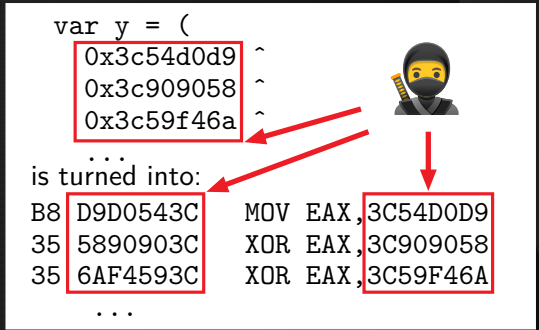
```
B8 D9D0543C    MOV EAX,3C54D0D9
35 5890903C    XOR EAX,3C909058
35 6AF4593C    XOR EAX,3C59F46A
    ...
```

EMOJI SHELLCODING IS A CODE-REUSE PROBLEM

Source: Trust me, bro.

- Return-Oriented Programming (Shacham et al., 2007):
  1. Scan target binary for reusable code snippets *(gadgets)*
  2. Build your shellcode with gadgets
  3. ???
  4. Profit
- JIT spraying (Blazakis, 2010): create your own gadgets by controlling JITted bytecode.

Easy when you control most of the output...



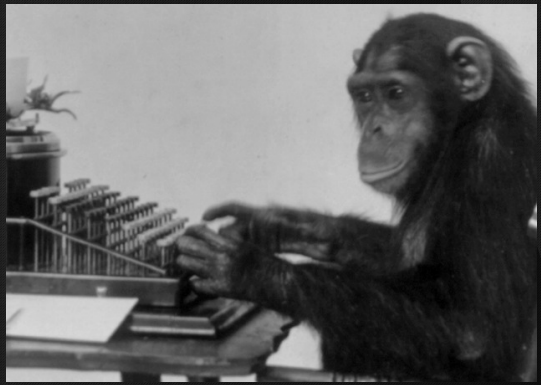
EMOJI SHELLCODING IS A CODE-REUSE PROBLEM

Great! How do I create gadgets from Emoji stream?

EMOJI SHELLCODING IS A CODE-REUSE PROBLEM

Great! How do I create gadgets from Emoji stream?

Remember infinite monkey theorem?



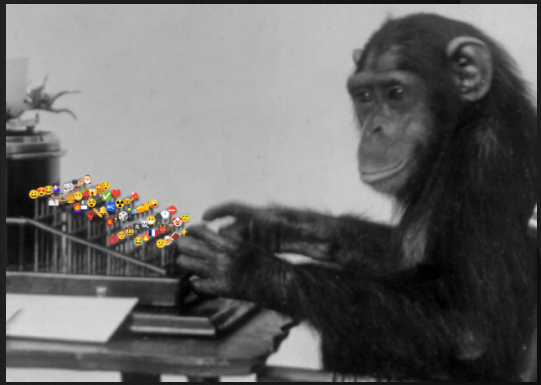
EMOJI SHELLCODING IS A CODE-REUSE PROBLEM

Great! How do I create gadgets from Emoji stream?

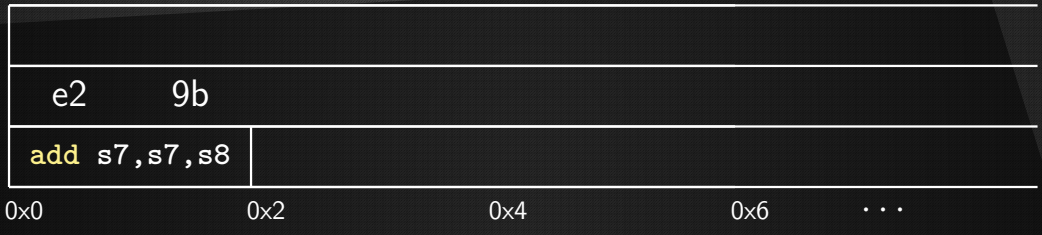
Remember infinite monkey theorem?

They get emoji keyboards now!

$\Rightarrow$ Need to invent a new algorithm.



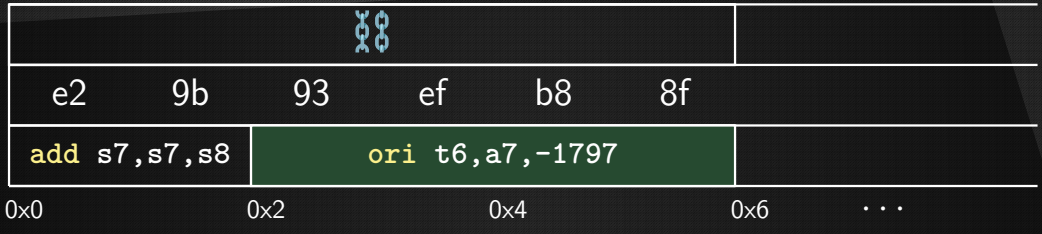
EMOJI GADGET BUILDING



EMOJI GADGET BUILDING (1ST CASE)



EMOJI GADGET BUILDING (1ST CASE)



EMOJI GADGET BUILDING (1ST CASE)



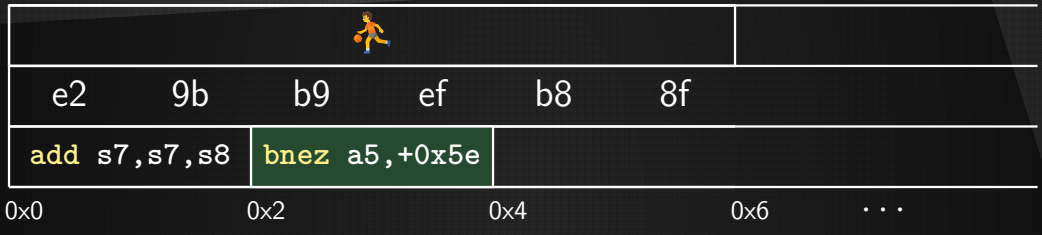

```
1 add s7,s7,s8; ori t6,a7,-1797
```

EMOJI GADGET BUILDING (2ND CASE)



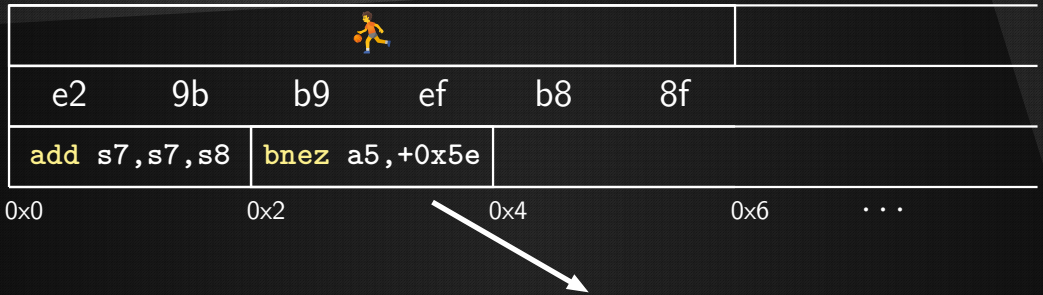

```
1 add s7,s7,s8; ori t6,a7,-1797
```

EMOJI GADGET BUILDING (2ND CASE)



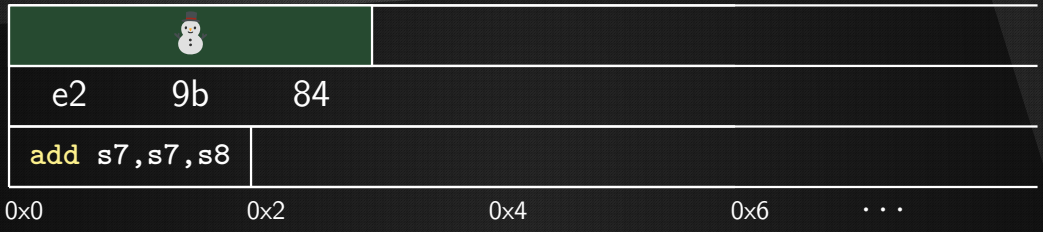

```
1 add s7,s7,s8; ori t6,a7,-1797
```

EMOJI GADGET BUILDING (2ND CASE)



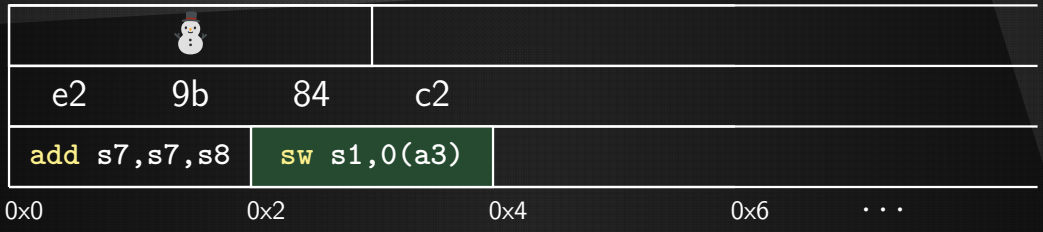

```
1 add s7,s7,s8; ori t6,a7,-1797
2 add s7,s7,s8; bnez a5,+0x5e; .byte 0xb8,0x8f
```

EMOJI GADGET BUILDING (3RD CASE)



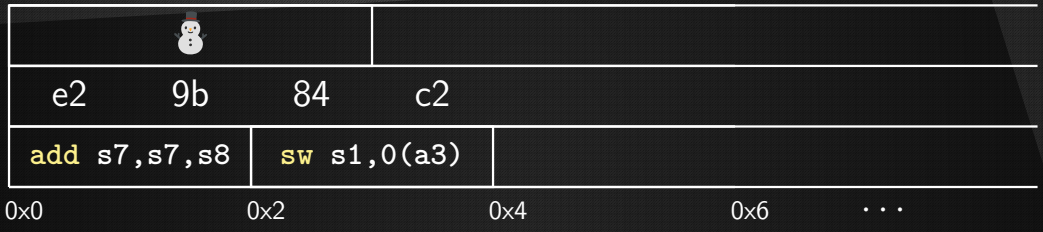

```
1 add s7,s7,s8; ori t6,a7,-1797
2 add s7,s7,s8; bnez a5,+0x5e; .byte 0xb8,0x8f
```

EMOJI GADGET BUILDING (3RD CASE)




```
1 add s7,s7,s8; ori t6,a7,-1797
2 add s7,s7,s8; bnez a5,+0x5e; .byte 0xb8,0x8f
```

EMOJI GADGET BUILDING (3RD CASE)



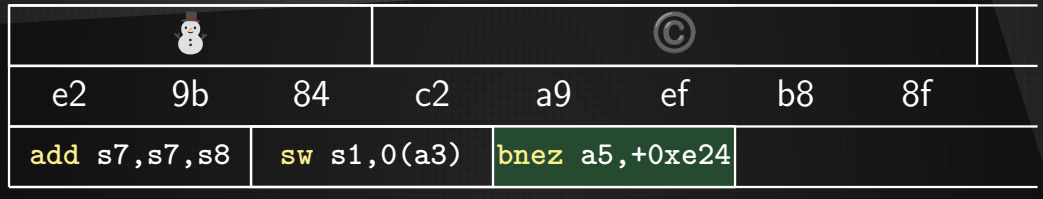

```
1 add s7,s7,s8; ori t6,a7,-1797
2 add s7,s7,s8; bnez a5,+0x5e; .byte 0xb8,0x8f
```

EMOJI GADGET BUILDING (3RD CASE)



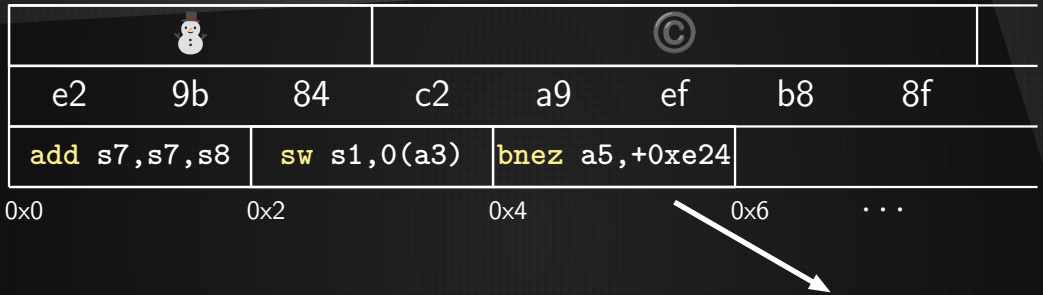

```
1 add s7,s7,s8; ori t6,a7,-1797
2 add s7,s7,s8; bnez a5,+0x5e; .byte 0xb8,0x8f
```

EMOJI GADGET BUILDING (3RD CASE)



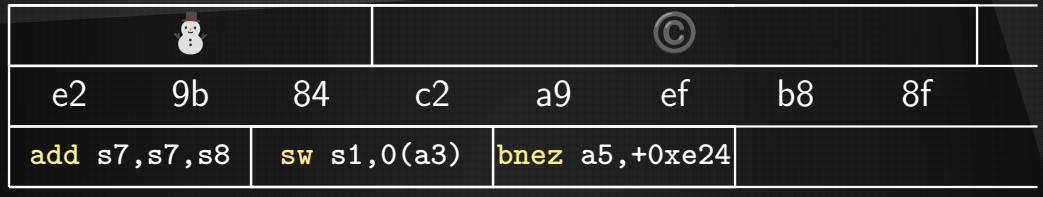

```
1 add s7,s7,s8; ori t6,a7,-1797
2 add s7,s7,s8; bnez a5,+0x5e; .byte 0xb8,0x8f
```

EMOJI GADGET BUILDING (3RD CASE)




```
1 add s7,s7,s8; ori t6,a7,-1797
2 add s7,s7,s8; bnez a5,+0x5e; .byte 0xb8,0x8f
3 add s7,s7,s8; sw s1,0(a3); bnez a5,+0xe24; .byte 0xb8,0x8f
```

EMOJI GADGET BUILDING




```
1 add s7,s7,s8; ori t6,a7,-1797
2 add s7,s7,s8; bnez a5,+0x5e; .byte 0xb8,0x8f
3 add s7,s7,s8; sw s1,0(a3); bnez a5,+0xe24; .byte 0xb8,0x8f
4 (and so on...)
```

THIS METHOD IS TOTALLY GENERIC

Algorithm follows derivations in the form $S \to a\ T$

This exactly describes a right-linear grammar!

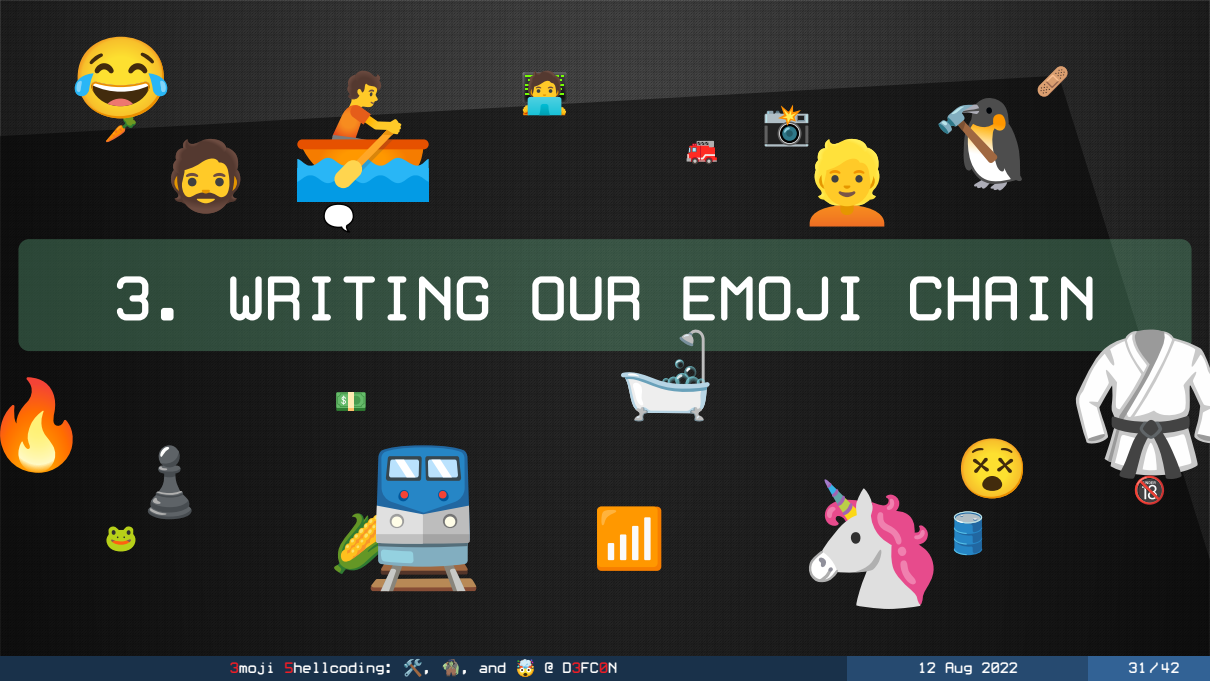
"Any regular expression* can be converted to a right-linear grammar."

*alphanumeric, email address, url, emojis, ... — A.R. Patel, 1971

(adapting the tool is left to the reader as exercise)

Anyway, back to emoji.

We have gadgets, but we still need to chain them!


3. WRITING OUR EMOJI CHAIN

POKÉDEX OF RISC-V EMOJI GADGETS

- >4000 instructions available in total
- Logic `and`, `add`, `sub`, ...
- Conditional and unconditional branches (forward & backward)
- Many registers: `a1357`, `s1359`, `t0246`, `gp`, `ra` (but no `sp` 😧)
- Very few immediates
- Lots of floating-point instructions

POKÉDEX OF RISC-V EMOJI GADGETS

- >4000 instructions available in total
- Logic `and`, `add`, `sub`, ...
- Conditional and unconditional branches (forward & backward)
- Many registers: `a1357`, `s1359`, `t0246`, `gp`, `ra` (but no `sp` 😧)
- Very few immediates
- Lots of floating-point instructions
- ... only one 'CSR' instruction: `csrrs ra,0xbf8,gp` 🤦

POKÉDEX OF RISC-V EMOJI GADGETS

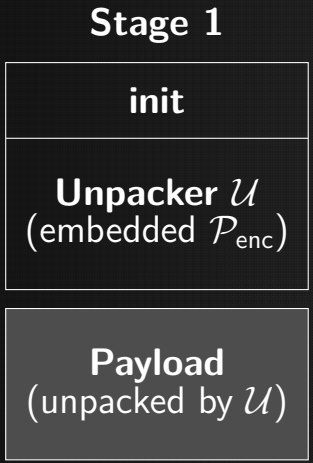
- >4000 instructions available in total
- Logic `and`, `add`, `sub`, ...
- Conditional and unconditional branches (forward & backward)
- Many registers: `a1357`, `s1359`, `t0246`, `gp`, `ra` (but no `sp` 😧)
- Very few immediates
- Lots of floating-point instructions
- ... only one 'CSR' instruction: `csrrs ra,0xbf8,gp` 🤦

A tiny bit of everything... looks like a yard sale!

FROM BASIC BLOCKS TO ARBITRARY CODE EXECUTION



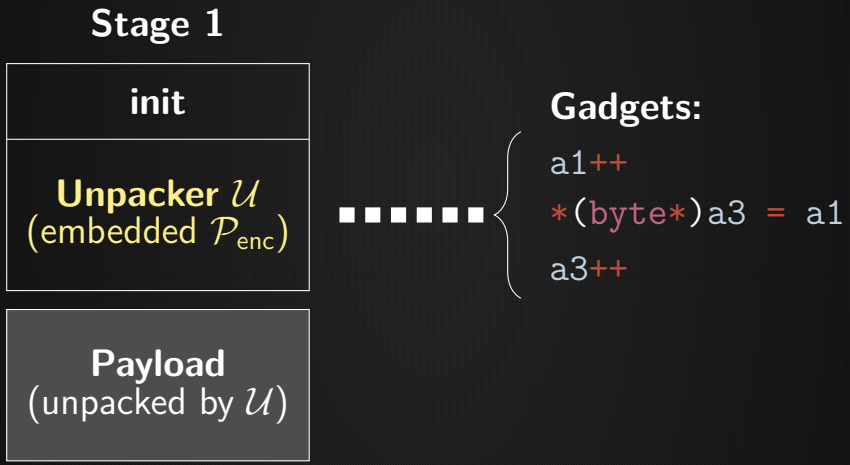
FROM BASIC BLOCKS TO ARBITRARY CODE EXECUTION



FROM BASIC BLOCKS TO ARBITRARY CODE EXECUTION



FROM BASIC BLOCKS TO ARBITRARY CODE EXECUTION



ARBITRARY PAYLOAD ENCODING

Initial Payload:

```
0x03
```

```
0x20
```

```
0x10
```

ARBITRARY PAYLOAD ENCODING

Initial Payload:	**Encoded $\mathcal{P}_{\text{enc}}$:**
0x03	
0x20	
0x10	

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:
0x03	`a1++; a1++; a1++;`
0x20	
0x10	

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:
0x03	`a1++; a1++; a1++;`
	`*(byte*)a3 = a1; a3++`
0x20	
0x10	

ARBITRARY PAYLOAD ENCODING

Initial Payload:	**Encoded $\mathcal{P}_{\text{enc}}$:**
0x03	`a1++; a1++; a1++;` `*(byte*)a3 = a1; a3++`
0x20	`a1++`$\ldots \times (\texttt{0x20} - \texttt{0x03}) = \texttt{0x1D}$
0x10	

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:
0x03	a1++; a1++; a1++;
	(byte)a3 = a1; a3++
0x20	a1++… $\times$ (0x20 − 0x03) = 0x1D
	(byte)a3 = a1; a3++
0x10	

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:
0x03	a1++; a1++; a1++; *(byte*)a3 = a1; a3++
0x20	a1++… $\times$ (0x20 − 0x03) = 0x1D *(byte*)a3 = a1; a3++
0x10	a1++… $\times$ (0x10 − 0x20) = 0xF0 *(byte*)a3 = a1

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:	Decoded Payload:
0x03	`a1++; a1++; a1++;`	
	`*(byte*)a3 = a1; a3++`	
0x20	`a1++`$\ldots \times(\texttt{0x20} - \texttt{0x03}) = \texttt{0x1D}$	
	`*(byte*)a3 = a1; a3++`	
0x10	`a1++`$\ldots \times(\texttt{0x10} - \texttt{0x20}) = \texttt{0xF0}$	
	`*(byte*)a3 = a1`	

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:	Decoded Payload:
0x03	`a1++; a1++; a1++;` `*(byte*)a3 = a1; a3++`	0x03
0x20	`a1++`…$\times(\texttt{0x20} - \texttt{0x03}) = \texttt{0x1D}$ `*(byte*)a3 = a1; a3++`	
0x10	`a1++`…$\times(\texttt{0x10} - \texttt{0x20}) = \texttt{0xF0}$ `*(byte*)a3 = a1`	

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:	Decoded Payload:
0x03	a1++; a1++; a1++; *(byte*)a3 = a1; a3++	0x03
0x20	a1++… $\times$ (0x20 − 0x03) = 0x1D *(byte*)a3 = a1; a3++	0x20
0x10	a1++… $\times$ (0x10 − 0x20) = 0xF0 *(byte*)a3 = a1	

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:	Decoded Payload:
0x03	a1++; a1++; a1++; *(byte*)a3 = a1; a3++	0x03
0x20	a1++… $\times$ (0x20 − 0x03) = 0x1D *(byte*)a3 = a1; a3++	0x20
0x10	a1++… $\times$ (0x10 − 0x20) = 0xF0 *(byte*)a3 = a1	0x10

ARBITRARY PAYLOAD ENCODING

Initial Payload:	Encoded $\mathcal{P}_{\text{enc}}$:	Decoded Payload:
0x03	a1++; a1++; a1++; *(byte*)a3 = a1; a3++	0x03
0x20	a1++… $\times$ (0x20 − 0x03) = 0x1D *(byte*)a3 = a1; a3++	0x20
0x10	a1++… $\times$ (0x10 − 0x20) = 0xF0 *(byte*)a3 = a1	0x10

GDB OVER BEAMER TIME!



GDB OVER BEAMER TIME!




```
add   s3,s3,s8
sw    a1,0(a3)     } *(byte*)a3 = a1
bnez  a5,+0x5a
...
add   s3,s3,s8
bnez  a3,+0x2      } a3++
add   a3,a3,t2
```

a1	0000 00AB
a3	0000 8000
t2	0000 0001
a5	B100 D5AC
s3	530F 25F8
s8	03C4 9ECC

...	00	...
...	0x8000	0x8001

GDB OVER BEAMER TIME!




```
add    s3,s3,s8
sw     a1,0(a3)      *(byte*)a3 = a1
bnez   a5,+0x5a
...
add    s3,s3,s8
bnez   a3,+0x2       a3++
add    a3,a3,t2
```

a1	0000 00AB
a3	0000 8000
t2	0000 0001
a5	B100 D5AC
s3	56D3 C4C4
s8	03C4 9ECC

···	00	···
···	0x8000	0x8001

GDB OVER BEAMER TIME!




```
add     s3,s3,s8
sw      a1,0(a3)      *(byte*)a3 = a1
bnez    a5,+0x5a
...
add     s3,s3,s8
bnez    a3,+0x2       a3++
add     a3,a3,t2
```

a1	0000 00AB
a3	0000 8000
t2	0000 0001
a5	B100 D5AC
s3	56D3 C4C4
s8	03C4 9ECC

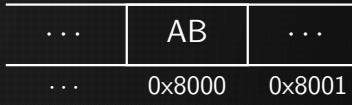


GDB OVER BEAMER TIME!




```
add    s3,s3,s8
sw     a1,0(a3)      *(byte*)a3 = a1
bnez   a5,+0x5a
...
add    s3,s3,s8
bnez   a3,+0x2       a3++
add    a3,a3,t2
```

a1	0000 00AB
a3	0000 8000
t2	0000 0001
a5	B100 D5AC
s3	56D3 C4C4
s8	03C4 9ECC



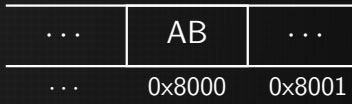
GDB OVER BEAMER TIME!

♌	©			...	❌	♋	
e2 99	8c c2	a9 ef	b8 8f	...	e2 9d	8c e2	99 8b
0x0	0x2	0x4	0x6	0x8 ···	0x62	0x64	0x66



```
add    s3,s3,s8
sw     a1,0(a3)      *(byte*)a3 = a1
bnez   a5,+0x5a
...
add    s3,s3,s8
bnez   a3,+0x2       a3++
add    a3,a3,t2
```

a1	0000 00AB
a3	0000 8000
t2	0000 0001
a5	B100 D5AC
s3	5A98 6390
s8	03C4 9ECC



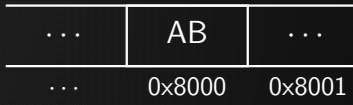
...	AB	...
...	0x8000	0x8001

GDB OVER BEAMER TIME!




```
add    s3,s3,s8
sw     a1,0(a3)      } *(byte*)a3 = a1
bnez   a5,+0x5a
...
add    s3,s3,s8
bnez   a3,+0x2       } a3++
add    a3,a3,t2
```

a1	0000 00AB
a3	0000 8000
t2	0000 0001
a5	B100 D5AC
s3	5A98 6390
s8	03C4 9ECC

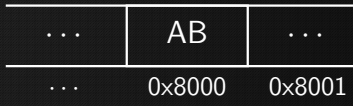


GDB OVER BEAMER TIME!




```
add   s3,s3,s8
sw    a1,0(a3)      *(byte*)a3 = a1
bnez  a5,+0x5a
...
add   s3,s3,s8
bnez  a3,+0x2       a3++
add   a3,a3,t2
```

a1	0000 00AB
a3	0000 8001
t2	0000 0001
a5	B100 D5AC
s3	5A98 6390
s8	03C4 9ECC

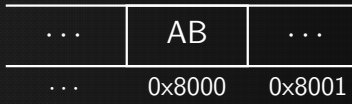


GDB OVER BEAMER TIME!




```
add   s3,s3,s8
sw    a1,0(a3)      } *(byte*)a3 = a1
bnez  a5,+0x5a
...
add   s3,s3,s8
bnez  a3,+0x2       } a3++
add   a3,a3,t2
```

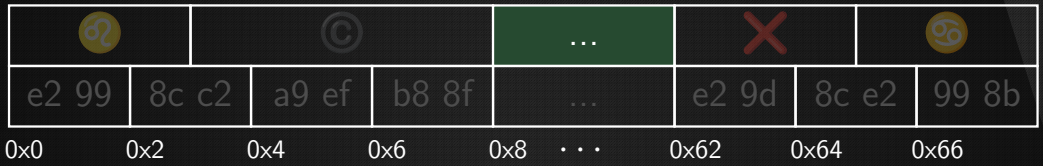
a1	0000 00AB
a3	0000 8001
t2	0000 0001
a5	B100 D5AC
s3	5A98 6390
s8	03C4 9ECC



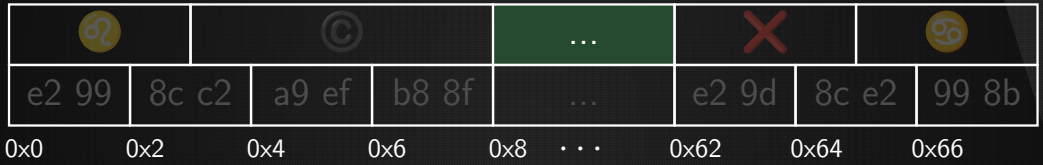
DEMO ON RV64GC SIFIVE UNLEASHED BOARD



FILLING THE VOIDS

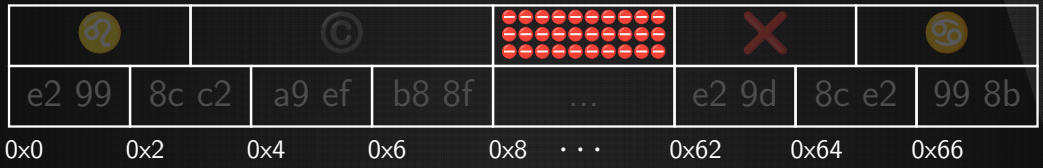


FILLING THE VOIDS



- Just solve a variant of subset sum (RTFM Wikipedia).

FILLING THE VOIDS



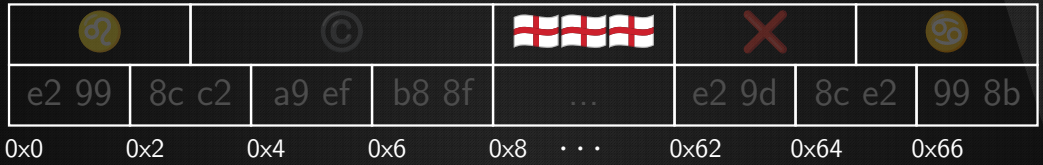
- Just solve a variant of subset sum (RTFM Wikipedia).
  - Trivial if we use only 3 and 4-byte emojis …

FILLING THE VOIDS



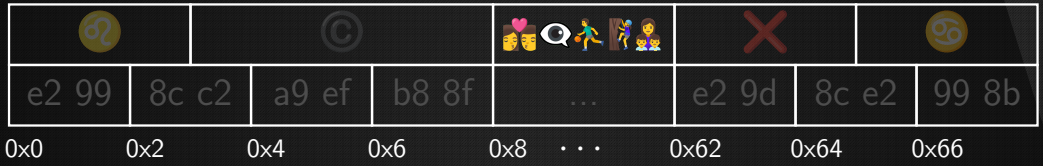
- Just solve a variant of subset sum (RTFM Wikipedia).
  - Trivial if we use only 3 and 4-byte emojis ...
  - ... harder if we want to minimize number of emojis
    $\Rightarrow$ dynamic programming (takes 4'13" to implement)

FILLING THE VOIDS



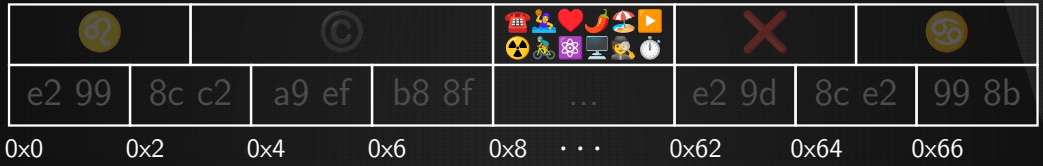
- Just solve a variant of subset sum (RTFM Wikipedia).
  - Trivial if we use only 3 and 4-byte emojis ...
  - ... harder if we want to minimize number of emojis
    $\Rightarrow$ dynamic programming (takes 4'13" to implement)
- Fantastic, we get polymorphism for free!

FILLING THE VOIDS



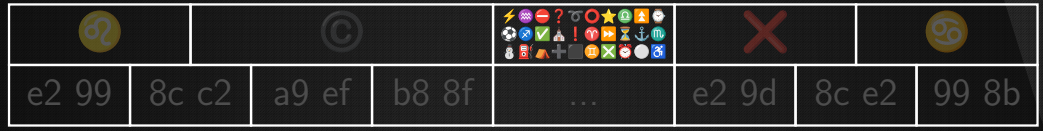
- Just solve a variant of subset sum (RTFM Wikipedia).
  - Trivial if we use only 3 and 4-byte emojis ...
  - ... harder if we want to minimize number of emojis
    $\Rightarrow$ dynamic programming (takes 4'13" to implement)
- Fantastic, we get polymorphism for free!

FILLING THE VOIDS



- Just solve a variant of subset sum (RTFM Wikipedia).
  - Trivial if we use only 3 and 4-byte emojis ...
  - ... harder if we want to minimize number of emojis
    $\Rightarrow$ dynamic programming (takes 4'13" to implement)
- Fantastic, we get polymorphism for free!

FILLING THE VOIDS



- Just solve a variant of subset sum (RTFM Wikipedia).
  - Trivial if we use only 3 and 4-byte emojis …
  - … harder if we want to minimize number of emojis
    ⇒ dynamic programming (takes 4'13" to implement)
- Fantastic, we get polymorphism for free!

TRIVIA

Can we do our nop-sled with the  emoji?

TRIVIA

Can we do our nop-sled with the  emoji?

✅

TRIVIA

Can we do our nop-sled with the 👊 emoji?

✅©

TRIVIA

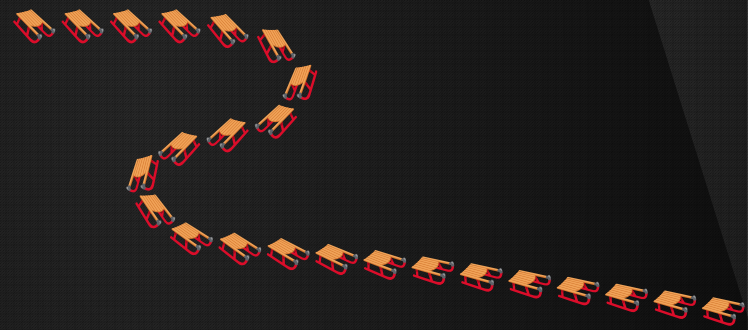
Can we do our nop-sled with the 🦴 emoji?

✅©⛔🙅⛔⛔⛔🙅⛔

TRIVIA

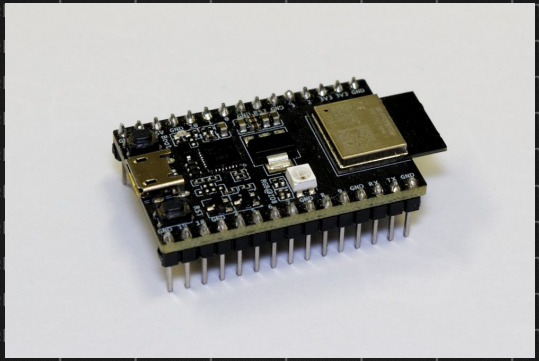
Can we do our nop-sled with the 👇 emoji?

✅©⛔🤷⛔⛔⛔🤷⛔



This is executable code!

DEMO ON RV32 ESPRESSIF ESP32-C3 BOARD



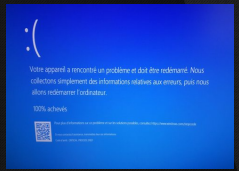
666. CONCLUSION


666. CONCLUSION

ACKNOWLEDGMENTS

The following were harmed during the making of this project:

- LuaLatex (segfault)
- TexStudio (memory leaks, crashes, ...)
- Acrobat Reader (major glitches)
- Various PDF readers (glitches)
- VLC (glitches on demo.mkv)
- Firefox (broke PDF.js persistently)
- xfce4-terminal (utter slowness)
- Noto Emoji (render issues)
- CUPS (crash)
- gcc (segfault)
- Windows (BSOD when compiling slides)



Emoji support **is** hard.

We went all the way to design a new code-reuse technique just to hack you with unsolicited “I 💖 U” texts.

Code and documentation on:

https://github.com/RischardV/emoji-shellcoding

Short link: bit.do/dc30

Your friendly neighbourhood hackers

hadrien.barral@ens.fr georges-axel.jaloyan@ens.fr