WHO WE ARE

- This is our third/second talk at D3FC0N.
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.

Jump using vuln

- NOP sled
- Payload
- Target’s memory
SHELLCODE 102
Not as convenient as you may think

- If treated as a string, no \x00
- If treated as input, no whitespace
- Other constraints (regex)
- Breaks easily (string escaping)
- Does not really look legit 🙄
Constrained shellcoding on x86 is easy:

<table>
<thead>
<tr>
<th>ASCII</th>
<th>IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>'P'</td>
<td>push %eax</td>
</tr>
<tr>
<td>'h'</td>
<td>push imm32</td>
</tr>
<tr>
<td>'X'</td>
<td>pop %eax</td>
</tr>
<tr>
<td>'a'</td>
<td>popa</td>
</tr>
<tr>
<td>'A'</td>
<td>inc %eax</td>
</tr>
<tr>
<td>'H'</td>
<td>dec %eax</td>
</tr>
<tr>
<td>'t'</td>
<td>je short disp8</td>
</tr>
<tr>
<td>'u'</td>
<td>jnz short disp8</td>
</tr>
<tr>
<td>'v'</td>
<td>jbe short disp8</td>
</tr>
<tr>
<td>'8'</td>
<td>cmp r/m8 r8</td>
</tr>
<tr>
<td>'0'</td>
<td>xor r/m8 r8</td>
</tr>
</tbody>
</table>
Constrained shellcoding on x86 is easy:

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

<table>
<thead>
<tr>
<th>ASCII</th>
<th>IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>'P'</td>
<td>push %eax</td>
</tr>
<tr>
<td>'h'</td>
<td>push imm32</td>
</tr>
<tr>
<td>'X'</td>
<td>pop %eax</td>
</tr>
<tr>
<td>'a'</td>
<td>popa</td>
</tr>
<tr>
<td>'A'</td>
<td>inc %eax</td>
</tr>
<tr>
<td>'H'</td>
<td>dec %eax</td>
</tr>
<tr>
<td>'t'</td>
<td>je short disp8</td>
</tr>
<tr>
<td>'u'</td>
<td>jnz short disp8</td>
</tr>
<tr>
<td>'v'</td>
<td>jbe short disp8</td>
</tr>
<tr>
<td>'8'</td>
<td>cmp r/m8 r8</td>
</tr>
<tr>
<td>'0'</td>
<td>xor r/m8 r8</td>
</tr>
</tbody>
</table>
Constrained shellcoding on x86 is easy:

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

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

<table>
<thead>
<tr>
<th>ASCII</th>
<th>IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>'P'</td>
<td>push %eax</td>
</tr>
<tr>
<td>'h'</td>
<td>push imm32</td>
</tr>
<tr>
<td>'X'</td>
<td>pop %eax</td>
</tr>
<tr>
<td>'a'</td>
<td>popa</td>
</tr>
<tr>
<td>'A'</td>
<td>inc %eax</td>
</tr>
<tr>
<td>'H'</td>
<td>dec %eax</td>
</tr>
<tr>
<td>'t'</td>
<td>je short disp8</td>
</tr>
<tr>
<td>'u'</td>
<td>jnz short disp8</td>
</tr>
<tr>
<td>'v'</td>
<td>jbe short disp8</td>
</tr>
<tr>
<td>'8'</td>
<td>cmp r/m8 r8</td>
</tr>
<tr>
<td>'0'</td>
<td>xor r/m8 r8</td>
</tr>
</tbody>
</table>
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
- **喰** for x86-64: just prefix `0x48 (= H)` to every instruction

<table>
<thead>
<tr>
<th>ASCII</th>
<th>IA32</th>
</tr>
</thead>
<tbody>
<tr>
<td>'P'</td>
<td>push %eax</td>
</tr>
<tr>
<td>'h'</td>
<td>push imm32</td>
</tr>
<tr>
<td>'X'</td>
<td>pop %eax</td>
</tr>
<tr>
<td>'a'</td>
<td>popa</td>
</tr>
<tr>
<td>'A'</td>
<td>inc %eax</td>
</tr>
<tr>
<td>'H'</td>
<td>dec %eax</td>
</tr>
<tr>
<td>'t'</td>
<td>je short disp8</td>
</tr>
<tr>
<td>'u'</td>
<td>jnz short disp8</td>
</tr>
<tr>
<td>'v'</td>
<td>jbe short disp8</td>
</tr>
<tr>
<td>'8'</td>
<td>cmp r/m8 r8</td>
</tr>
<tr>
<td>'0'</td>
<td>xor r/m8 r8</td>
</tr>
</tbody>
</table>
CRAZY PEOPLE EVEN MADE ENGLISH SHELLCODES

Mason et al., CCS, 2009
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:

■ Easy to compile to one-instruction set computers (e.g. Movfuscator)

Cons:
THE "COMPILATION" WAY

Idea:
- Compile assembly code to a constrained instruction set

Pros:
- Easy to compile to one-instruction set computers (e.g. Movfuscator)

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 😢
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:
- Interpreted code is toothless... 😞
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, ...)

3moji Shellcoding: 🕰️, 📗, and 🌐 @ DEFCON
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 execute payload 😸
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)

Can you emoji shellcode?

Nope, because XYZ!

Shit XYZ is wrong!

Hadrien

I now haz to write emoji shellcode

WTF is this?

With all those bugs, how does it even work?

Let's clean this shit together

@BloodyTangerine

Hadrien

Georges-Axel

Georges-Axel

Hadrien
2. INSPECTOR GADGET
WHAT IS AN EMOJI?

[Image of a hand holding a smartphone with the text: I ❤️ you]
WHAT IS AN EMOJI?

(or more likely a mess 😳 ...)

I 😘 you
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, 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, ま, 🃏, 🌸, 😊,
WHAT IS AN EMOJI?

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

In Unicode, you’ll find:

A, ま, ₄, ４, 😊, 🛠, 🌟,

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.
WHAT IS AN EMOJI?

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

In Unicode, you’ll find:

A, ま, ぁ, ♠, 😊, 🌒, 🌠, 🌠, 🌠, 🌠, 🌠,

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 bytes. And they add new emojis every year! Currently at Unicode version 14.
WHAT IS AN EMOJI?

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

In Unicode, you’ll find:

A, ま, Ⅲ, ♣, 😊, 🖤, 🎄, 🪖, 🌠,

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.
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. And they add new emojis every year! Currently at Unicode version 14.
WHAT IS AN EMOJI?

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

In Unicode, you’ll find:

A, ま, ♦️, 😊, 🇫🇷, 🌙, and 😔

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.
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. And they add new emojis every year! Currently at Unicode version 14.
WHAT IS AN EMOJI?

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

In Unicode, you’ll find:

A, ま, ｧ, 苡, 🌻, 🌺, 🫣, 🌹.

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, ｂ, 卡, 😊, 🌺, 🌴, 🌞, 🌌,

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, ま, あ, け, 😊,  '&#x1047;+', 𒈙, 卜

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

RISC-V®
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*

*: 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*

*: Conditions may apply
EXCUSE ME, DO YOU HAVE 2 MINUTES TO TALK ABOUT

RISC-V

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

*:Conditions may apply
HOW CAN I EXECUTE MY EMOJI?
HOW CAN I EXECUTE MY EMOJI?

- ✔ Alphanumeric x86: ‘A’: **inc %eax**
  
  0x41

- 🟢 Alphanumeric ARM&RISCV: ‘4A0s’:
  
  csrc mip,sp
  
  0x34413073

- 🐻 Emoji RISCV:
  
  0xE29D93
HOW CAN I EXECUTE MY EMOJI?

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

We are stuck!

---

emoji Shellcoding: 🔨, 🚀, and 🤖 @ DEFCON

12 Aug 2022

21/42
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’: csrb 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 😭
  `0xE29D93E29D93E29D93E29D93...`

We are stuck!
THE INTUITION...

auipc ra,621055
THE INTUITION...

97 F0 9F 97

auipc ra,621055
THE INTUITION...

97  F0  9F  97

auipc  ra,621055
THE INTUITION...

F0 9F 86 97  F0 9F 97

auipc ra,621055
THE INTUITION...

F0 9F 86 97 F0 9F 97

auipc ra,621055
Jump

<none>

F0 9F 86 97

F0 9F 97

auipc ra,621055
Jump

F0 9F 86

E2 9D 97

F0 9F 97

auipc ra,621055

THE INTUITION...
Jump

<none>

\textbf{F0} \textbf{9F} \textbf{86} \textbf{97} \textbf{F0} \textbf{9F} \textbf{97}

\textbf{E2} \textbf{9D}

\textbf{add s11,s8}

\textbf{auipc ra,621055}
THE INTUITION...

Jump

F0 9F 86 E2 9D

add s11, s8

auipc ra, 621055

Jump
THE INTUITION...

Jump

<none>

F0 9F 86 E2 9D
add s11, s8
auipc ra, 621055

F0 9F 97
93 EF B8 8F

!
Jump

\[
\text{add s11,s8} \quad \text{auipc ra,621055} \quad \text{ori t6,a7,-1797}
\]

\[\text{F0 9F 86 E2 9D} \quad 97 \quad \text{F0 9F 97} \quad \text{93 EF B8 8F}\]
THE INTUITION...

Jump

add s11, s8

auipc ra, 621055

ori t6, a7, -1797

bnez a5, +28
THE INTUITION...

Jump

97

<none>

add s11,s8

auipc ra,621055

ori t6,a7,-1797

bnez a5,+28

Jump

3moji Shellcoding: 🌡️, ⌚️, and 🎨 @ DEFCON

12 Aug 2022

22/42
Jump

<none>

add s11, s8

auipc ra, 621055

ori t6, a7, -1797

bnez a5, +28
How do I generate all gadgets?
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.

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

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

```javascript
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

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?

⇒ He got emoji keyboard now!

⇒ Need to invent a new algorithm.
<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Symbol</th>
<th>Assembly Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>e2</td>
<td>add s7, s7, s8</td>
</tr>
<tr>
<td>0x2</td>
<td>9b</td>
<td>bnez a5, +0x5e;</td>
</tr>
<tr>
<td>0x4</td>
<td></td>
<td>.byte 0xb8, 0x8f</td>
</tr>
<tr>
<td>0x6</td>
<td></td>
<td>bnez a5, +0xe24;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.byte 0xb8, 0x8f</td>
</tr>
<tr>
<td>(and so on..)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This is a snippet of computer code, specifically assembly language, demonstrating the use of hexadecimal numbers and assembly instructions for arithmetic operations and branching conditions.
**EMOJI GADGET BUILDING (1ST CASE)**

<table>
<thead>
<tr>
<th></th>
<th>e2</th>
<th>9b</th>
<th>93</th>
<th>ef</th>
<th>b8</th>
<th>8f</th>
<th>0x0</th>
<th>0x2</th>
<th>0x4</th>
<th>0x6</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add s7,s7,s8; ori t6,a7,-1797; bnez a5,+0x5e; .byte 0xb8,0x8f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(and so on...)

---

3moji Shellcoding: 🎁, 🚀, and 🤖 @ DEFCON

12 Aug 2022

26/42
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td>9b</td>
<td>93</td>
<td>ef</td>
<td>b8</td>
<td>8f</td>
<td></td>
</tr>
<tr>
<td><strong>add s7,s7,s8</strong></td>
<td><strong>ori t6,a7,-1797</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0x2</td>
<td>0x4</td>
<td>0x6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1 add s7,s7,s8; ori t6,a7,-1797
1 add s7,s7,s8; ori t6,a7,-1797
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)

<table>
<thead>
<tr>
<th>e2</th>
<th>9b</th>
<th>84</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add s7, s7, s8</td>
<td>sw s1, 0(a3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Code Sequence

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)

<table>
<thead>
<tr>
<th>e2</th>
<th>9b</th>
<th>84</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add s7, s7, s8</td>
<td>sw s1, 0(a3)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 0x0 | 0x2 | 0x4 | 0x6 | ⋮ |

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)**

<table>
<thead>
<tr>
<th>e2</th>
<th>9b</th>
<th>84</th>
<th>c2</th>
<th>a9</th>
<th>ef</th>
<th>b8</th>
<th>8f</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add s7,s7,s8</strong></td>
<td><strong>sw s1,0(a3)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 0x0 | 0x2 | 0x4 | 0x6 | ...

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)

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td>9b</td>
<td>84</td>
<td>c2</td>
<td>a9</td>
<td>ef</td>
<td>b8</td>
<td>8f</td>
<td></td>
</tr>
<tr>
<td>add s7, s7, s8</td>
<td>sw s1, 0(a3)</td>
<td>bnez a5, +0xe24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x0 0x2 0x4 0x6 ... 

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

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td>9b</td>
<td>84</td>
<td>c2</td>
<td>a9</td>
<td>ef</td>
<td>b8</td>
<td>8f</td>
<td></td>
</tr>
<tr>
<td>add s7,s7,s8</td>
<td>sw s1,0(a3)</td>
<td>bnez a5,+0xe24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
0x0 0x2 0x4 0x6 ... 
```

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...)
Algorithm follows derivations in the form $S \rightarrow (\text{emoji} | \text{emoji} | \text{emoji} | \ldots) \ T$

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)
- ... only one ‘CSR’ instruction: `csrrs ra,0xbf8,gp`

- 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)
- ... only one ‘CSR’ instruction: csrrs ra,0xbf8,gp

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

- Many registers: a1357, s1359, t0246, gp, ra (but no sp 😕)
- Very few immediates
- Lots of floating-point instructions
FROM BASIC BLOCKS TO ARBITRARY CODE EXECUTION

Stage 1

<table>
<thead>
<tr>
<th>init</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpacker $U$ (embedded $P_{enc}$)</td>
</tr>
<tr>
<td>Payload (unpacked by $U$)</td>
</tr>
</tbody>
</table>
FROM BASIC BLOCKS TO ARBITRARY CODE EXECUTION

Stage 1

<table>
<thead>
<tr>
<th>init</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unpacker</strong> $U$ (embedded $P_{enc}$)</td>
</tr>
<tr>
<td><strong>Payload</strong> (unpacked by $U$)</td>
</tr>
</tbody>
</table>
Stage 1

<table>
<thead>
<tr>
<th>init</th>
<th>Gadgets:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unpacker U</strong></td>
<td>a1++</td>
</tr>
<tr>
<td>(embedded $P_{enc}$)</td>
<td>* (byte*)a3 = a1</td>
</tr>
<tr>
<td>Payload</td>
<td>a3++</td>
</tr>
<tr>
<td>(unpacked by U)</td>
<td></td>
</tr>
</tbody>
</table>
ARBITRARY PAYLOAD ENCODING

Initial Payload:

0x03 0x20 0x10

Encoded Payload:

a1++; a1++; a1++;
*(byte*)a3 = a1; a3++;
a1++;

× (0x20 − 0x03) = 0x1D

a1++; a1++; a1++;
*(byte*)a3 = a1;

a1++; a1++; a1++;
× (0x10 − 0x20) = 0xF0

Decoded Payload:

0x03 0x20 0x10

OSM emoji Shellcoding: 🖤, 🌆, and 😄 @ DEFCON 12 Aug 2022 34/42
ARBITRARY PAYLOAD ENCODING

Initial Payload: 0x03 0x20 0x10

Encoded $P_{enc}$:

- \[ a1++; a1++; a1++; *(byte*)a3 = a1; a3++ \]
- \[ a1++; *(byte*)a3 = a1; a3++ \]
- \[ \times (0x20 - 0x03) = 0x1D \]
- \[ \times (0x10 - 0x20) = 0xFO \]

Decoded Payload: 0x03 0x20 0x10
ARBITRARY PAYLOAD ENCODING

**Initial Payload:**
- 0x03
- 0x20
- 0x10

**Encoded** $P_{\text{enc}}$:
- `a1++; a1++; a1++;`

**Decoded Payload:**
- 0x03
- 0x20
- 0x10
<table>
<thead>
<tr>
<th>Initial Payload</th>
<th>Encoded $P_{enc}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td><code>a1++; a1++; a1++;</code></td>
</tr>
<tr>
<td></td>
<td><code>*(byte*)a3 = a1; a3++</code></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
</tbody>
</table>
**ARBITRARY PAYLOAD ENCODING**

<table>
<thead>
<tr>
<th>Initial Payload</th>
<th>Encoded $P_{enc}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>a1++; a1++; a1++;</td>
</tr>
<tr>
<td></td>
<td><em>(byte</em>)a3 = a1; a3++</td>
</tr>
<tr>
<td>0x20</td>
<td>a1++ ... × (0x20 − 0x03) = 0x1D</td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
</tbody>
</table>
## ARBITRARY PAYLOAD ENCODING

<table>
<thead>
<tr>
<th>Initial Payload:</th>
<th>Encoded $P_{enc}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>$a1++; a1++; a1++;$</td>
</tr>
<tr>
<td></td>
<td><em>(byte</em>)$a3 = a1; a3++*</td>
</tr>
<tr>
<td>0x20</td>
<td>$a1++ \cdots \times (0x20 - 0x03) = 0x1D$</td>
</tr>
<tr>
<td></td>
<td><em>(byte</em>)$a3 = a1; a3++*</td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
</tbody>
</table>
ARBITRARY PAYLOAD ENCODING

Initial Payload: 0x03 0x20 0x10

Encoded $P_{enc}$:

- $a1++; a1++; a1++;$
- $*(\text{byte}*)a3 = a1; a3++$
- $a1++ \ldots \times (0x20 - 0x03) = 0x1D$
- $*(\text{byte}*)a3 = a1; a3++$
- $a1++ \ldots \times (0x10 - 0x20) = 0xF0$
- $*(\text{byte}*)a3 = a1$

Decoded Payload: 0x03 0x20 0x10
ARBITRARY PAYLOAD ENCODING

Initial Payload: 0x03 0x20 0x10

Encoded $P_{\text{enc}}$:

- $a_1++; a_1++; a_1++;$
- $*(\text{byte}*)a_3 = a_1; a_3++$
- $a_1++(0x20 - 0x03) = 0x1D$
- $*(\text{byte}*)a_3 = a_1; a_3++$
- $a_1++(0x10 - 0x20) = 0xF0$
- $*(\text{byte}*)a_3 = a_1$

Decoded Payload: 0x03 0x20 0x10
**Initial Payload:**

- 0x03
- 0x20
- 0x10

**Encoded Payload:**

- `a1++; a1++; a1++;`  
  `*(byte*)a3 = a1; a3++`
- `a1++ ... × (0x20 − 0x03) = 0x1D`  
  `*(byte*)a3 = a1; a3++`
- `a1++ ... × (0x10 − 0x20) = 0xFO`  
  `*(byte*)a3 = a1`

**Decoded Payload:**

- 0x03
ARBITRARY PAYLOAD ENCODING

**Initial Payload:**
0x03
0x20
0x10

**Encoded \( P_{enc} \):**

```
a1++; a1++; a1++;
*(byte*)a3 = a1; a3++
a1++ ...
\times (0x20 - 0x03) = 0x1D
*(byte*)a3 = a1; a3++
a1++ ...
\times (0x10 - 0x20) = 0xFF
*(byte*)a3 = a1
```

**Decoded Payload:**

0x03
0x20
0x10
### ARBITRARY PAYLOAD ENCODING

<table>
<thead>
<tr>
<th>Initial Payload</th>
<th>Encoded $P_{\text{enc}}$:</th>
<th>Decoded Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>(a1++; a1++; a1++;)</td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td>((\text{byte}*)a3 = a1; a3++)</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>(a1++ \times (0x20 - 0x03) = 0x1D)</td>
<td>0x20</td>
</tr>
<tr>
<td></td>
<td>((\text{byte}*)a3 = a1; a3++)</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>(a1++ \times (0x10 - 0x20) = 0xF0)</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td>((\text{byte}*)a3 = a1)</td>
<td></td>
</tr>
</tbody>
</table>
### ARBITRARY PAYLOAD ENCODING

<table>
<thead>
<tr>
<th>Initial Payload:</th>
<th>Encoded $P_{enc}$</th>
<th>Decoded Payload:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x03</td>
<td>$a1++; a1++; a1++;$</td>
<td>0x03</td>
</tr>
<tr>
<td></td>
<td><em>(byte</em>)$a3 = a1; a3++$</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>$a1++; \ldots \times (0x20 - 0x03) = 0x1D$</td>
<td>0x20</td>
</tr>
<tr>
<td></td>
<td><em>(byte</em>)$a3 = a1; a3++$</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>$a1++; \ldots \times (0x10 - 0x20) = 0xFF0$</td>
<td>0x10</td>
</tr>
<tr>
<td></td>
<td><em>(byte</em>)$a3 = a1$</td>
<td></td>
</tr>
</tbody>
</table>
GDB OVER BEAMER TIME!

<table>
<thead>
<tr>
<th>0x2</th>
<th>8c</th>
<th>c2</th>
<th>a9</th>
<th>ef</th>
<th>b8</th>
<th>8f</th>
<th>...</th>
<th>e2</th>
<th>9d</th>
<th>8c</th>
<th>e2</th>
<th>99</th>
<th>8b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x2</td>
<td>0x4</td>
<td>0x6</td>
<td>0x8</td>
<td>...</td>
<td>0x62</td>
<td>0x64</td>
<td>0x66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
add s3, s3, s8
```
```
sw a1, 0(a3)
```
```
bnez a5, +0x5a
```
```
add a3, a3, t2
```
```
... *(byte*)a3 = a1
```
```
bnez a3, +0x2
```
```
add a3, a3, t2
```
```
... 00 ... 0x8000 0x8001
```
```
... 0x8000 0x8001
```
```
a1 0000 00AB
a3 0000 8000
t2 0000 0001
a5 B100 D5AC
s3 530F 25F8
s8 03C4 9ECC
```

emoji Shellcoding: 🛠️, 🔧, and 🔴 D3FC0N
### GDB OVER BEAMER TIME!

<table>
<thead>
<tr>
<th>0x0</th>
<th>0x2</th>
<th>0x4</th>
<th>0x6</th>
<th>0x8</th>
<th>...</th>
<th>0x62</th>
<th>0x64</th>
<th>0x66</th>
</tr>
</thead>
<tbody>
<tr>
<td>e2 99</td>
<td>8c c2</td>
<td>a9 ef</td>
<td>b8 8f</td>
<td></td>
<td></td>
<td>e2 9d</td>
<td>8c e2</td>
<td>99 8b</td>
</tr>
</tbody>
</table>

### Assembly Code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>add s3, s3, s8</td>
<td>a1</td>
<td>0000 00AB</td>
</tr>
<tr>
<td>sw a1, 0(a3)</td>
<td>a3</td>
<td>0000 8000</td>
</tr>
<tr>
<td>bnez a5, +0x5a</td>
<td>t2</td>
<td>0000 0001</td>
</tr>
<tr>
<td></td>
<td>a5</td>
<td>B100 D5AC</td>
</tr>
<tr>
<td></td>
<td>s3</td>
<td>56D3 C4C4</td>
</tr>
<tr>
<td></td>
<td>s8</td>
<td>03C4 9ECC</td>
</tr>
<tr>
<td>add s3, s3, s8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnez a3, +0x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add a3, a3, t2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Register Values

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a3</td>
<td>0000 8000</td>
</tr>
<tr>
<td>t2</td>
<td>0000 0001</td>
</tr>
<tr>
<td>a5</td>
<td>B100 D5AC</td>
</tr>
<tr>
<td>s3</td>
<td>56D3 C4C4</td>
</tr>
<tr>
<td>s8</td>
<td>03C4 9ECC</td>
</tr>
</tbody>
</table>

### Memory Locations

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>0x8001</td>
</tr>
</tbody>
</table>

---

*Note: The diagram and table are visual representations of the assembly code's execution.*
### GDB OVER BEAMER TIME!

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e2 99</td>
<td>8c c2</td>
<td>a9 ef</td>
<td>b8 8f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0x2</td>
<td>0x4</td>
<td>0x6</td>
<td>0x8</td>
<td></td>
</tr>
</tbody>
</table>

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

*(byte*)a3 = a1

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>0000 00AB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td>0000 8000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>0000 0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a5</td>
<td>B100 D5AC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>56D3 C4C4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s8</td>
<td>03C4 9ECC</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
*(byte*)a3 = a1
```

```
add s3, s3, s8
```

```
bnez a3, +0x2
```

```
add a3, a3, t2
```

---

### Memory Locations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8000</td>
<td>0x8001</td>
</tr>
</tbody>
</table>

---
GDB OVER BEAMER TIME!

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

<table>
<thead>
<tr>
<th>a2</th>
<th>©</th>
<th>©</th>
<th>...</th>
<th>e2</th>
<th>8c</th>
<th>c2</th>
<th>a9</th>
<th>ef</th>
<th>b8</th>
<th>8f</th>
<th>...</th>
<th>e2</th>
<th>9d</th>
<th>8c</th>
<th>e2</th>
<th>99</th>
<th>8b</th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td>99</td>
<td>8c</td>
<td>c2</td>
<td>a9</td>
<td>ef</td>
<td>b8</td>
<td>8f</td>
<td>...</td>
<td>e2</td>
<td>9d</td>
<td>8c</td>
<td>e2</td>
<td>99</td>
<td>8b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0x0 0x2 0x4 0x6 0x8 0x62 0x64 0x66

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

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

... | AB | ...

... 0x8000 0x8001
```
**GDB OVER BEAMER TIME!**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td>99</td>
<td>8c</td>
<td>c2</td>
<td>a9</td>
<td>ef</td>
<td>b8</td>
</tr>
<tr>
<td>0x0</td>
<td>0x2</td>
<td>0x4</td>
<td>0x6</td>
<td>0x8</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>add</td>
<td>s3 , s3, s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw</td>
<td>a1 , 0(a3)</td>
<td><em>(byte</em>)a3 = a1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td>a5 , +0x5a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>s3 , s3, s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bnez</td>
<td>a3 , +0x2</td>
<td>a3++</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>a3 , a3, t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
add s3, s3, s8
sw a1, 0(a3)
bnez a5, +0x5a

*(byte*)a3 = a1
```
**GDB OVER BEAMER TIME!**

<table>
<thead>
<tr>
<th>(e2)</th>
<th>(8c)</th>
<th>(c2)</th>
<th>(a9)</th>
<th>(ef)</th>
<th>(b8)</th>
<th>(8f)</th>
<th>...</th>
<th>(e2)</th>
<th>(9d)</th>
<th>(8c)</th>
<th>(e2)</th>
<th>(99)</th>
<th>(8b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x2</td>
<td>0x4</td>
<td>0x6</td>
<td>0x8</td>
<td>...</td>
<td>0x62</td>
<td>0x64</td>
<td>0x66</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{add} & \quad \text{s3, s3, s8} \\
\text{sw} & \quad \text{a1, 0(a3)} \\
\text{bnez} & \quad \text{a5, +0x5a} \\
\text{...} & \quad \text{...} \\
\text{add} & \quad \text{s3, s3, s8} \\
\text{bnez} & \quad \text{a3, +0x2} \\
\text{add} & \quad \text{a3, a3, t2}
\end{align*}
\]

\[\begin{align*}
\text{a3++} & \\
\text{*(byte*)a3} & = \text{a1}
\end{align*}\]

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

\[
\begin{align*}
\ldots \quad \ldots & \\
\ldots & 0x8000 \quad 0x8001
\end{align*}
\]
### GDB OVER BEAMER TIME!

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e2</td>
<td>99</td>
<td>8c</td>
<td>c2</td>
<td>a9</td>
<td>ef</td>
<td>b8</td>
<td>8f</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td>0x2</td>
<td>0x4</td>
<td>0x6</td>
<td>0x8</td>
<td></td>
<td></td>
<td></td>
<td>0x62</td>
<td>0x64</td>
</tr>
</tbody>
</table>

```
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  
```
**GDB OVER BEAMER TIME!**

<table>
<thead>
<tr>
<th>a2</th>
<th>a9</th>
<th>b8</th>
<th>c2</th>
<th>e2</th>
<th>8c</th>
<th>8f</th>
<th>99</th>
<th>8b</th>
<th>0x0</th>
<th>0x2</th>
<th>0x4</th>
<th>0x6</th>
<th>0x8</th>
<th>0x62</th>
<th>0x64</th>
<th>0x66</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>s3 , s3 , s8</td>
<td><em>(byte</em>)a3 = a1</td>
<td>sw</td>
<td>a1 , 0(a3)</td>
<td>bnez</td>
<td>a5 , +0x5a</td>
<td>...</td>
<td>add</td>
<td>s3 , s3 , s8</td>
<td>a3++</td>
<td>add</td>
<td>a3 , a3 , t2</td>
<td>...</td>
<td>...</td>
<td>AB</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

... 0x8000 0x8001...
DEMO ON RV64GC SIFIVE UNLEASHED BOARD
Just solve a variant of subset sum (RTFM Wikipedia).

Trivial as we have 3 and 4-byte emojis ...

... harder if we want to have fun.

e.g. minimize number of emojis

⇒ dynamic programming.

Fantastic, we get polymorphism for free!

3moji Shellcoding: 🔨, 🧪, and 🤖 @ DEFCON
12 Aug 2022
Just solve a variant of subset sum (RTFM Wikipedia).
FILLING THE VOIDS

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

- Trivial as we have 3 and 4-byte emojis...
FILLING THE VOIDS

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

- Trivial as we have 3 and 4-byte emojis ...
- ... harder if we want to have fun.
Just solve a variant of subset sum (RTFM Wikipedia).

- Trivial as we have 3 and 4-byte emojis ...
- ... harder if we want to have fun.
- e.g. minimize number of emojis ⇒ dynamic programming.
Just solve a variant of subset sum (RTFM Wikipedia).

- Trivial as we have 3 and 4-byte emojis ...
- ... harder if we want to have fun.
- e.g. minimize number of emojis ⇒ dynamic programming.

Fantastic, we get polymorphism for free!
Just solve a variant of subset sum (RTFM Wikipedia).

- Trivial as we have 3 and 4-byte emojis ...
- ... harder if we want to have fun.
- e.g. minimize number of emojis ⇒ dynamic programming.

Fantastic, we get polymorphism for free!
Just solve a variant of subset sum (RTFM Wikipedia).
  - Trivial as we have 3 and 4-byte emojis ...
  - ... harder if we want to have fun.
  - e.g. minimize number of emojis ⇒ dynamic programming.

Fantastic, we get polymorphism for free!
Can we do our nop-sled with the 🛸 emoji?
Can we do our nop-sled with the 🛫 emoji?
Can we do our nop-sled with the 🛷 emoji?
TRIVIA

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