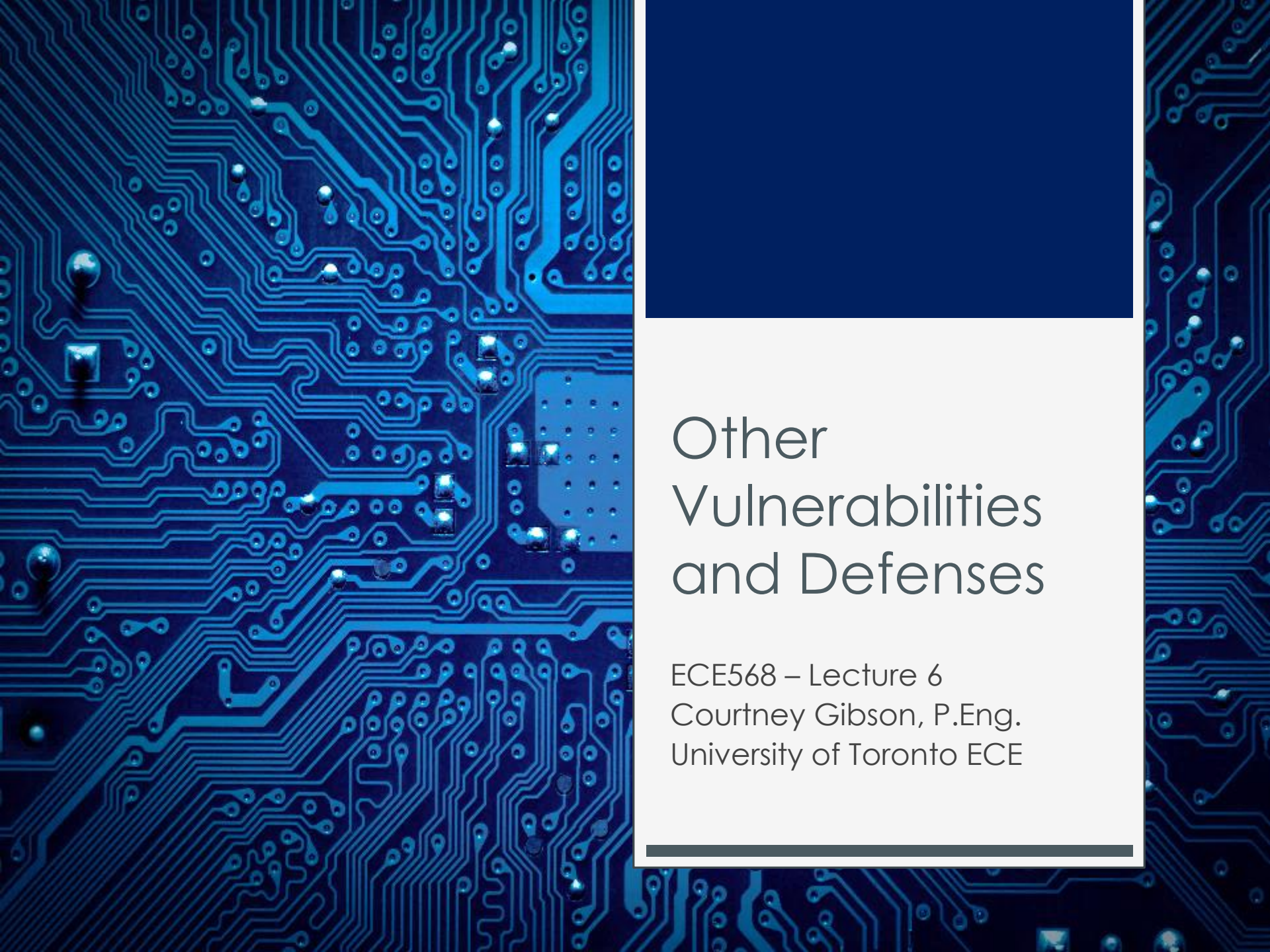


Other Vulnerabilities and Defenses

ECE568 – Lecture 6

Courtney Gibson, P.Eng.
University of Toronto ECE

Outline

Other Common Vulnerabilities

- Return-into-libc attacks
- Function pointer overwrite
- PLT/GOT overwrite
- Integer overflow
- Bad bounds check
- Argument overwrite

Defenses

- Stackguard stack smashing defense
- Address space layout randomization
- Non-executable pages

Other Common Vulnerabilities

Function pointers, dynamic linking, integer overflows, bad bounds checking, argument overwrite

Attacks without Code Injection

Until now, the attacks have involved overwriting the return address to point to injected code

- Is it possible to exploit a program without injecting code?
- What does it mean to exploit a program without injecting code?

An exploit can occur if an attacker can cause unintended program execution or unintended data modification

Return into **libc**

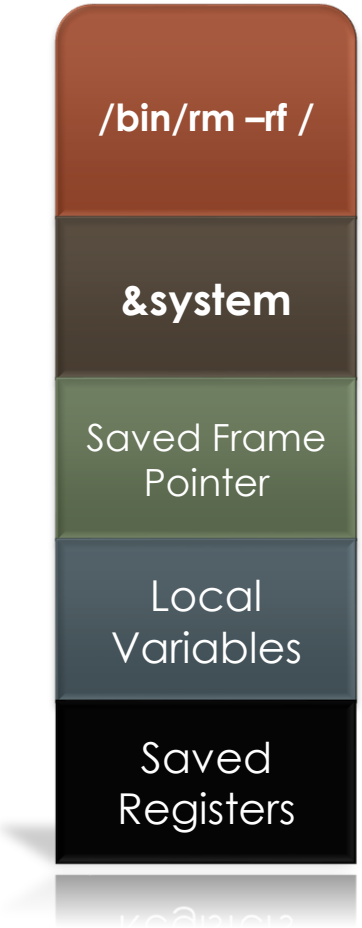
- One exploit method that doesn't require code injection is to use code already present (pre-64-bit)
- Many **libc** functions have code useful to the attacker
  - e.g., the **`system`** library call looks a lot like shell code:

```
int system(const char *string);


// system() executes a command specified in string
// by calling /bin/sh -c string
```

Return into **libc**

Rather than inject shellcode, attacker:

- Changes the return address to point to start of the **`system`** function
- Injects a stack frame on the stack
- Just before return, **`sp`** points to &system (the orig return address)
- On return, **`system`** executes and expects its arguments at the top of the stack
- Argument contains the string the attacker wants to execute



Attacks without Overwriting the Return Address

Until now, attacks have overwritten return address. Are there other exploit possibilities?

- Function pointer overwrite
- Global offset table overwrite

Function Pointers

A function pointer is a variable that can be dereferenced to call a function:

```
int foo(int arg1) {
  ...
}

int (*fp)(int arg1);   /* define a function pointer */
fp = &foo;     /* assign addr of foo to the pointer */
fp(6);                  /* call foo via the pointer */
```

An adversary can try to overwrite a function pointer

Function Pointer Overwrite

- Common in object-oriented languages (*e.g.*, C++)
- Function pointers are often used in C also
  - Allows mimicing polymorphic features of OO languages (*e.g.*, qsort)
  - Used to support dynamically loaded libraries
  - Very common in OS kernels, where the kernel has to run with different modules or drivers without recompilation
  - Also common in other programs that use modules such as web servers, etc.
- Sometimes a buffer will be beside a function pointer rather a return address
  - By overwriting the function pointer, the attacker can cause execution to be redirected when the program calls the function pointer

Dynamic Linking

Program code needs to call functions such as **`printf`** in dynamic libraries

- These libraries are normally linked into the program at run time, at arbitrary locations, by a dynamic linker
- Typically, both the caller of a library function and the function itself are compiled to be **position independent**
- We need to map the position independent function call to the absolute location of the function's code in the library
  - The dynamic linker performs this mapping
  - It uses two tables: the **Procedure Linkage Table** (PLT) and the **Global Offset Table** (GOT)

PLT/GOT

GOT is a table of pointers to functions:

- Contains the absolute memory location of each of the dynamically-loaded library functions

PLT is a table of code entries:

- One per each library function called by program
  - For example, sprintf@plt
- Somewhat similar to a switch statement
- Each code entry invokes the corresponding function pointer in GOT
  - For example, sprintf@PLT code may invoke "jmp GOT[k]", where k corresponds to the sprintf function index in GOT

Dynamic Linking Mechanism

All calls to dynamic libraries jump to PLT:

- The first time the function is called, the runtime linker is invoked to load the library
- The runtime linker updates the GOT entry, based on where the library is loaded
- Further calls invoke the function in the loaded library via the updated GOT entry
- The PLT/GOT contain commonly used library functions such as **`printf, fopen, fclose`**, etc.

PLT/GOT Overwrites

Suppose that an attacker is only able to overwrite a single chosen address location with a chosen value

- Then a good option is to overwrite a GOT function pointer

A binary utility like **`objdump -x`** allows dissassembling an executable

- It provides the location of these structures
- PLT/GOT always appear at a **known** location

Return-Oriented Programming

- Based on carefully-selected sequences of instructions, located at the end of existing functions (called “gadgets”)

```
…
mov   $0xb,%ebx
ret

…
mov $0x0,-8(%ebp)
ret
```

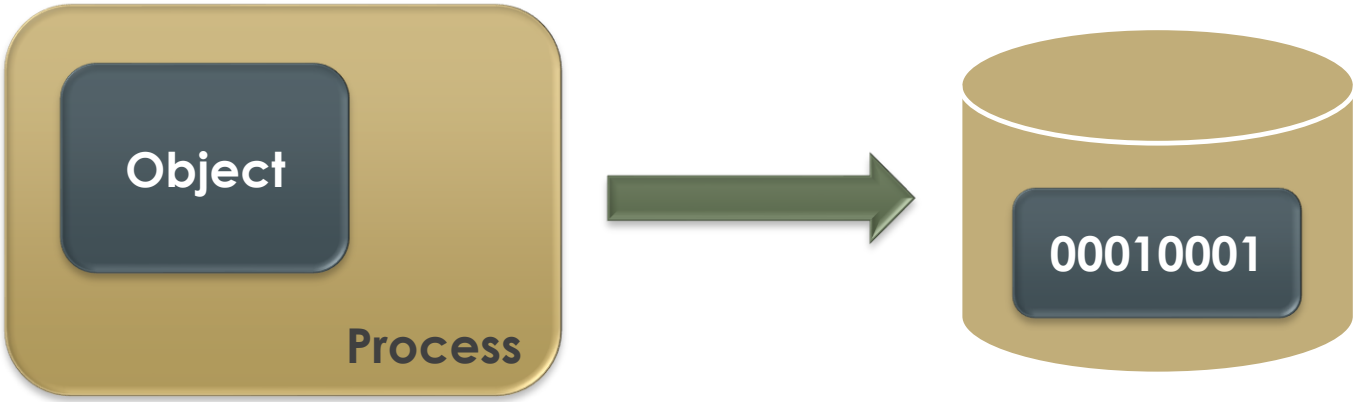
```
…
movl $0x0,%edx
ret

…
movl %ebx,%eax
ret
```

ROP

- Overflow the stack, placing a sequence of return addresses that correspond to the sequence of gadgets

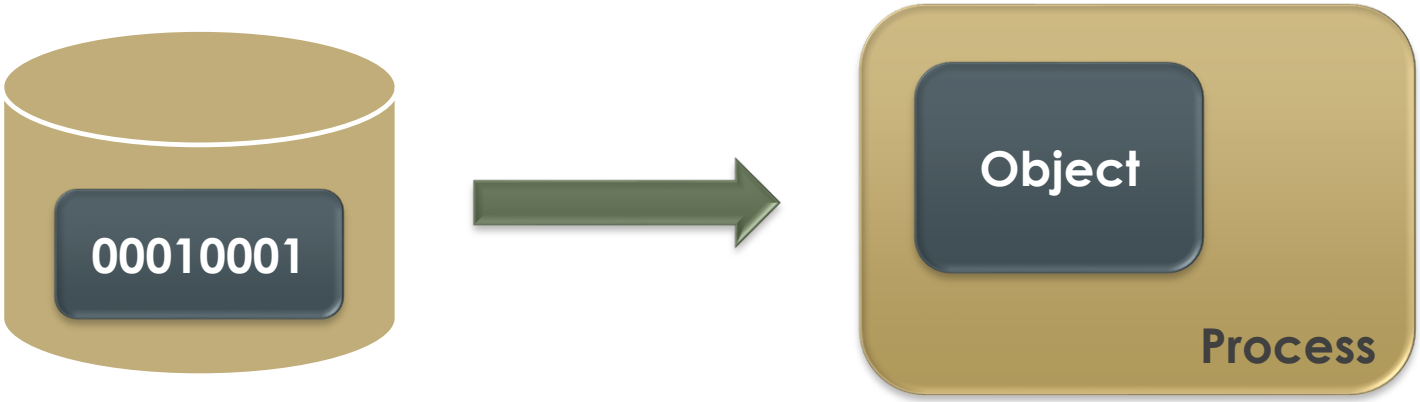
Deserialization Vulnerabilities

- **Serialization**
  - Process of translating objects into a format that can be stored or transmitted over a network

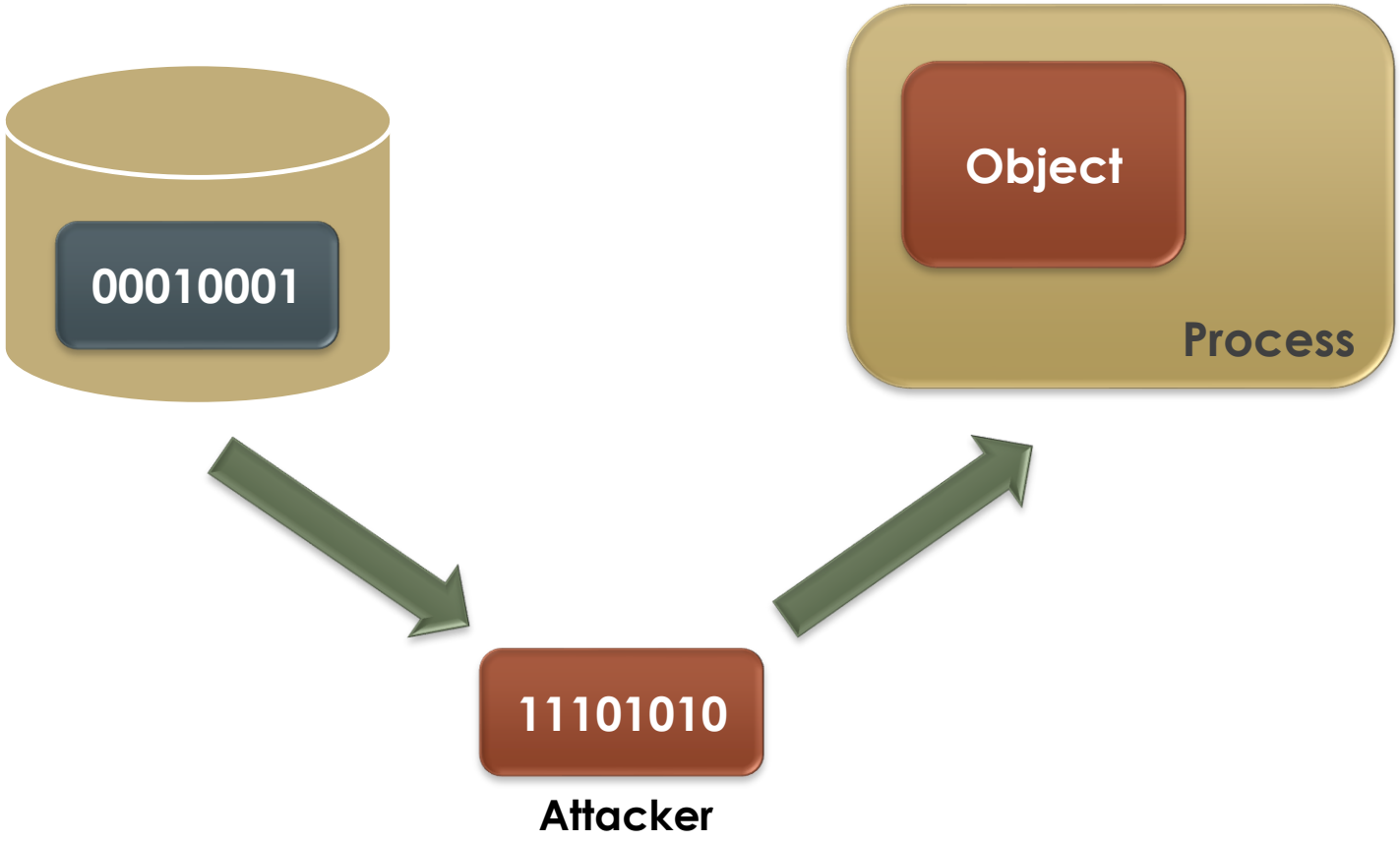


Deserialization Vulnerabilities

- **De-serialization**
  - Reconstructs the object in memory
  - Should not result in code execution (ideally)
  - However, a number of common libraries have vulnerabilities that can be exploited



Deserialization Vulnerabilities



Deserialization Vulnerabilities

- Numerous languages and toolkits
  - Java, Python, Scala, Websphere, JBOSS, WebLogic…
- For more information:
  - https://github.com/frohoff/ysoserial.git

Integer Overflows

A server processes packets of variable size:

- The first 2 bytes of the packet store the size of the packet to be processed
- Only packets of size <= 512 bytes should be processed
- What's wrong with the code?
- **Hint:** the third arg of memcpy is unsigned int

```
char* processNext(char* strm)
{
  char buf[512];
  short len = *(short*) strm;
  strm += sizeof(len);
  if (len <= 512) {
    memcpy(buf, strm, len);
    process(buf);
    return strm + len;
  } else {
    return -1;
  }
}
```

Bad Bounds Check

What's wrong with this bounds check?

```
/* Linux 2.4.5/drivers/char/drm/i810_dma.c */
/* [copy arg from user space into d] */
if(copy_from_user(&d, arg, sizeof(arg)))
  return -EFAULT;
if(d.idx > dma->buf_count)
  return -EINVAL;
buf = dma->buflist[d.idx];
copy_from_user(buf_priv->virtual, d.address, d.used);
```

Allows reading arbitrary memory locations

- Similar vulnerabilities have led to remote code execution in the past
  - e.g., the do_brk() function in the Linux 2.4.22 kernel

Argument Overwrite

Instead of changing the execution of a program (i.e. control flow), an attacker can cause unintended data modification

- For example, an attacker can hijack a program by overwriting the argument of a sensitive function such as **exec**

```
char buf[128] = “my_program” ;
char vulnerable[32];
        …
exec(buf);
```

The attacker can corrupt the argument **buf** by overflowing **vulnerable** and have the program execute something else

- Note that the program execution has not changed!

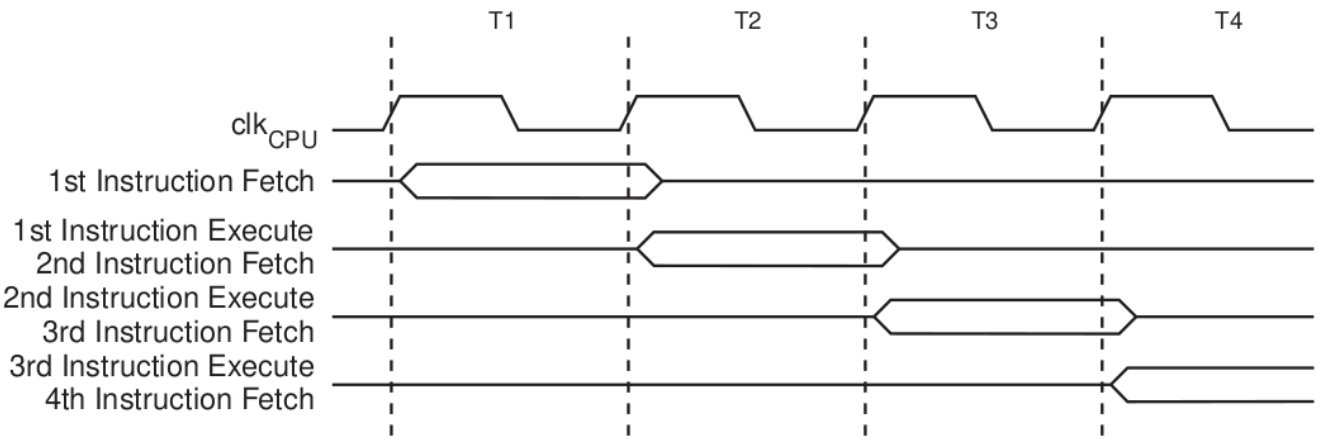
IoT

Hardware / Software Vulnerabilities


Welcome Home
Texecom
PROX
6759

Fault Injection Attacks

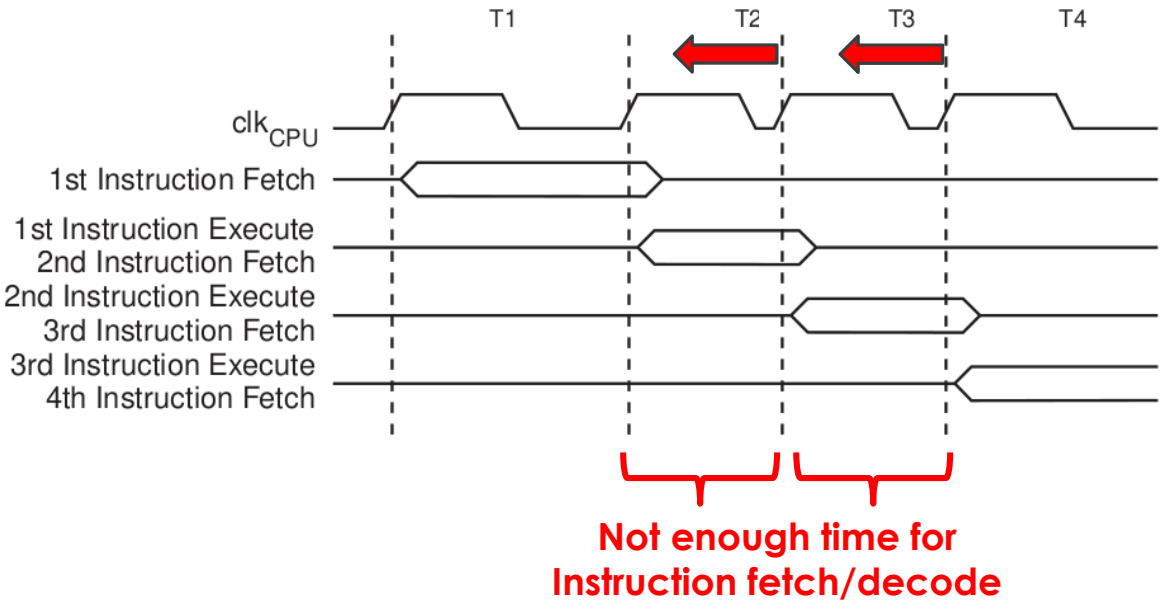
Observation: Proper CPU operation depends on stable power and clock inputs.



ATMega328P Instruction Pipeline

Clock Glitching

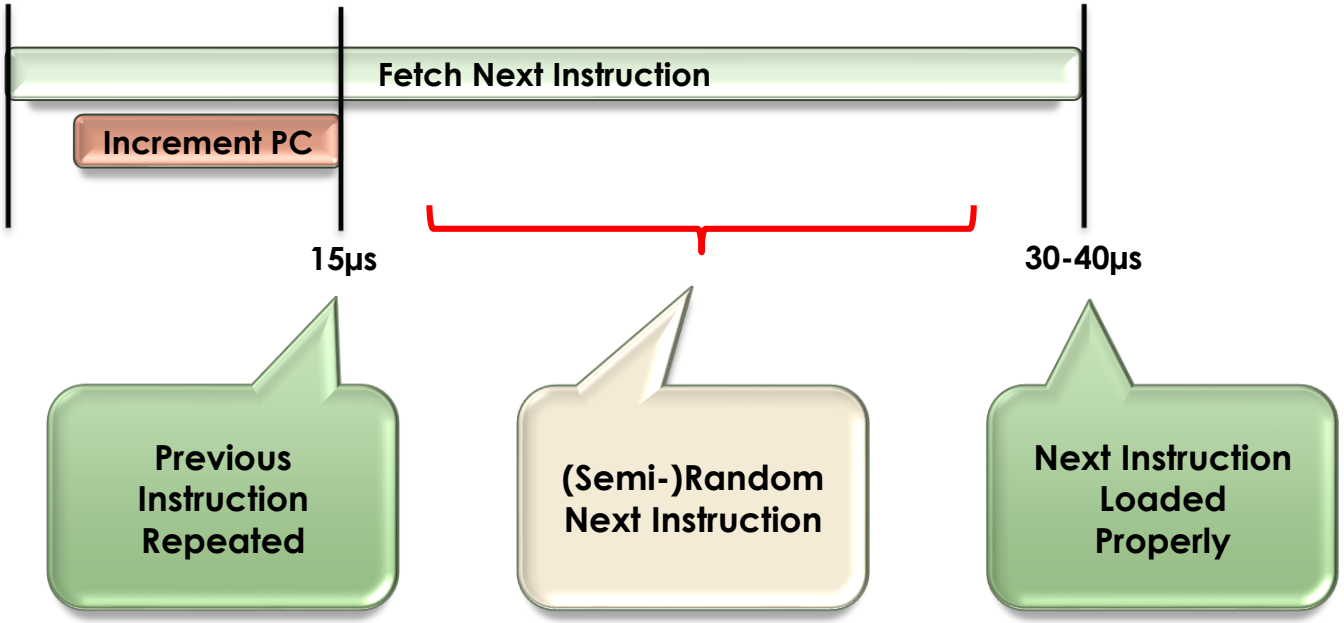
What happens if we introduce a very brief, rapid series of clock pulses? (*e.g.*, 30-40x the normal clock speed?)



If the glitch duration is:

1. **Longer** than the time to increment the Program Counter; and,
2. **Shorter** than the instruction fetch time

then we can start to see a special case: either **instruction skipping** or **instruction corruption**.



Instruction Skipping

```
…
if ( validPassword == true )
{
        startUserShell();
        …
}
```

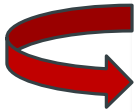
```
…
LOAD        R1, &validPassword
TEST        R1, 0
JMPEQ       +0x100
CALL        startUserShell
…
```

Instruction Skipping

```
…
LOAD        R1, &validPassword
TEST        R1, 0
JMPEQ       +100
CALL        startUserShell
…
```

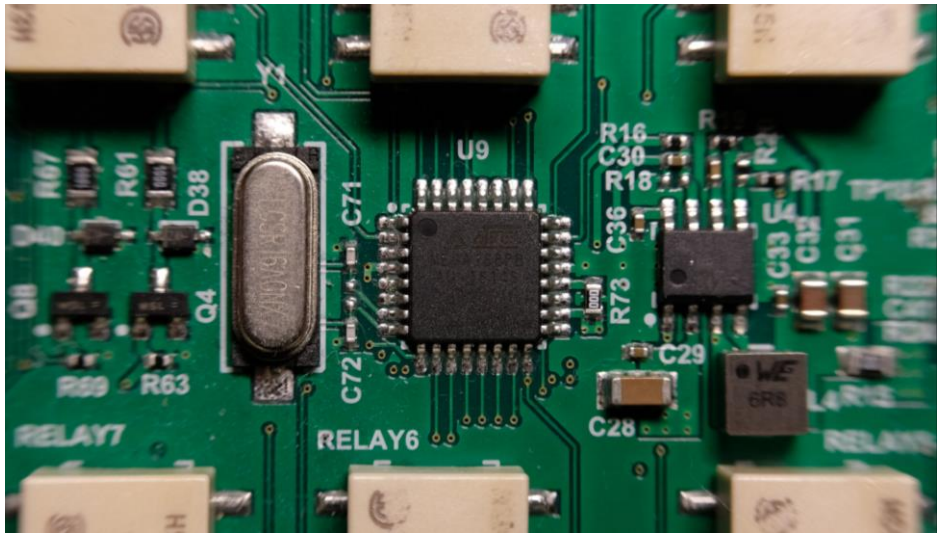
```
…
LOAD        R1, &validPassword
TEST        R1, 0
TEST        R1, 0
CALL        startUserShell
…
```

Repeated Instruction



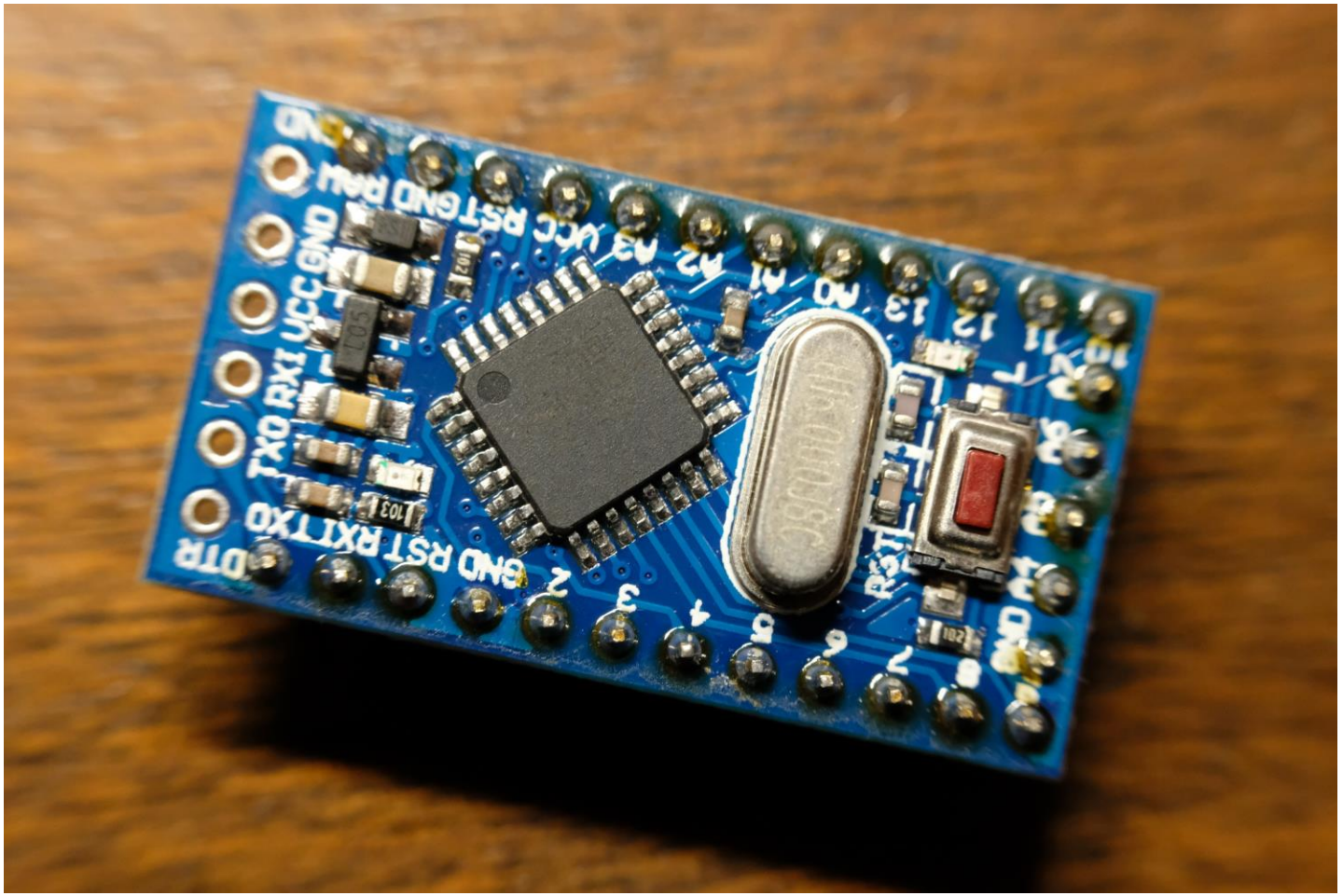
Demo

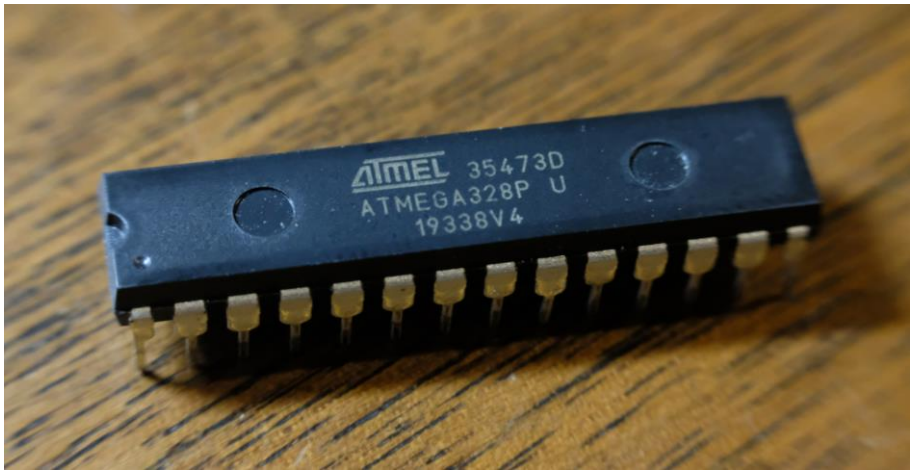
Clock and Power Glitching



ATMega328P Microcontroller

- Low-powered microprocessor + flash memory
- One application, no traditional operating system
- Common in **IoT** (home automation) and **embedded** (industrial control) applications

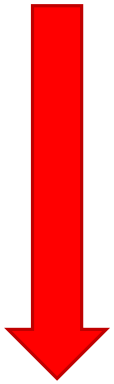

GND
TXO RXI VCC GND
GND RST RXI TXO
DTR
RST



Device Under Test (DUT)

- Our "victim" device will be the 28-pin version of the (easier to solder, but otherwise identical)
- Loaded with a test application that simulates a secure device with a login shell

```
login: root
Password: password

Login incorrect.

login: root
Password: s3cr3tP4ssw0rd&:-)@#

Last login: Wed Dec 31 12:34:56 2025 from 127.0.0.1
root@iotvictim:~#
```

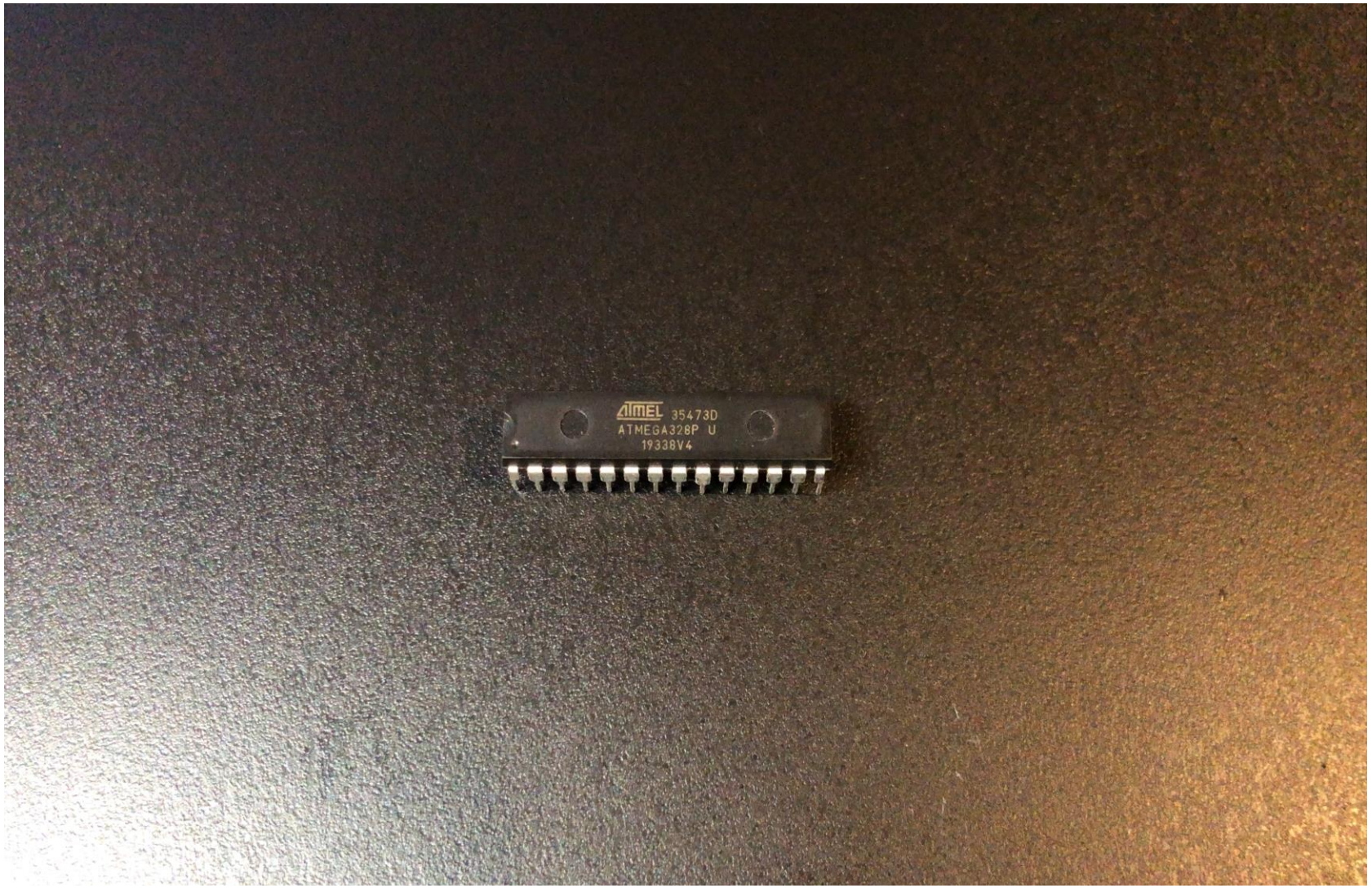


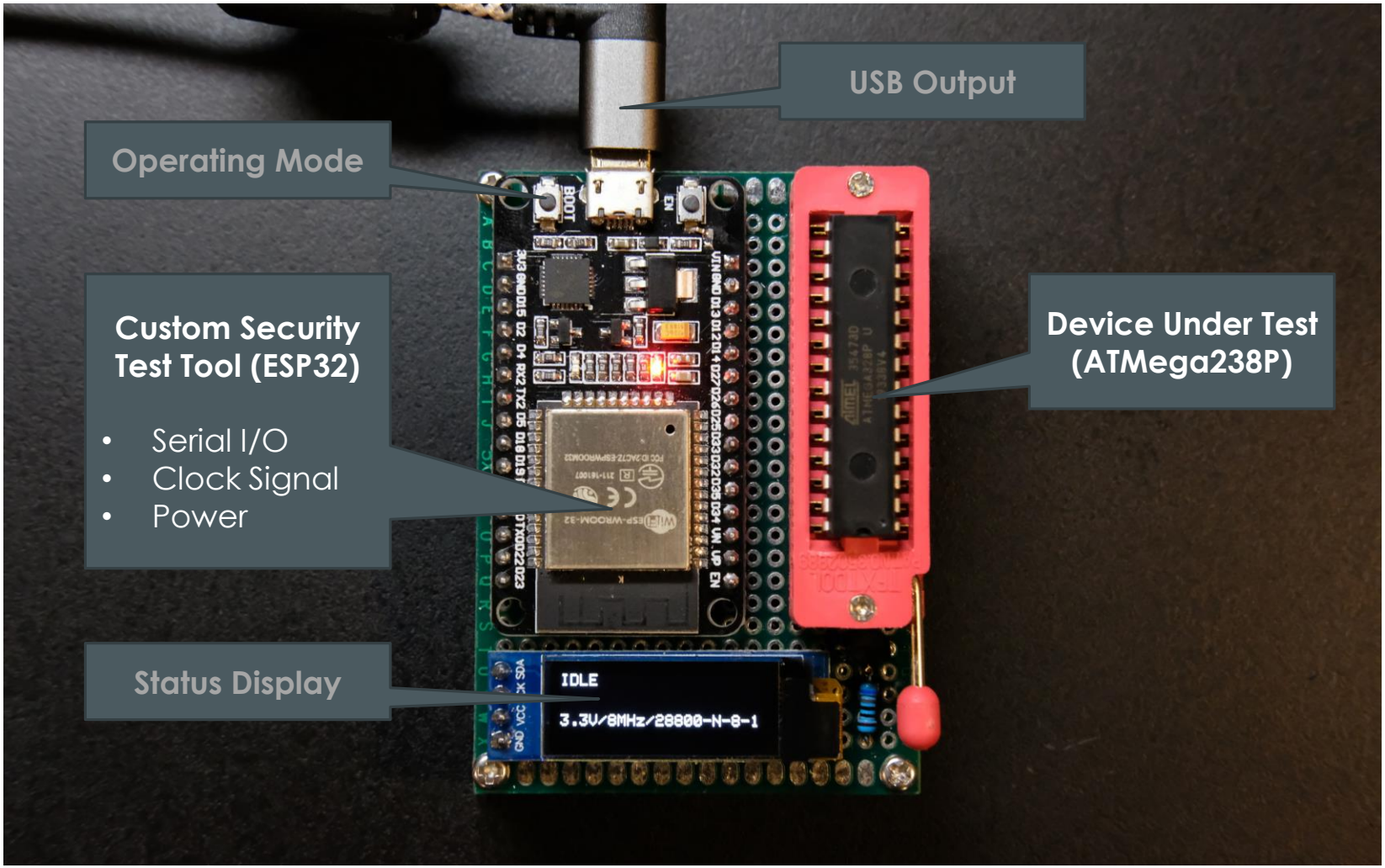
```
…
bool passwordIsValid =
        !strcmp("s3cr3tP4ssw0rd&:-)@#", buffer);

if ( !usernameIsValid || !passwordIsValid ) {

        // Login incorrect
        Serial.println("\n\nLogin incorrect.");
        return;
}

// Login correct
…
```

```
…
bool passwordIsValid =
       !strcmp("s3cr3tP4ssw0rd&:-)@#", buffer);

if ( !usernameIsValid || !passwordIsValid ) {

       // Login incorrect
       Serial.println("\n\nLogin incorrect.");
       return;
}

// Login correct
…
```

GOAL: SKIP THIS!

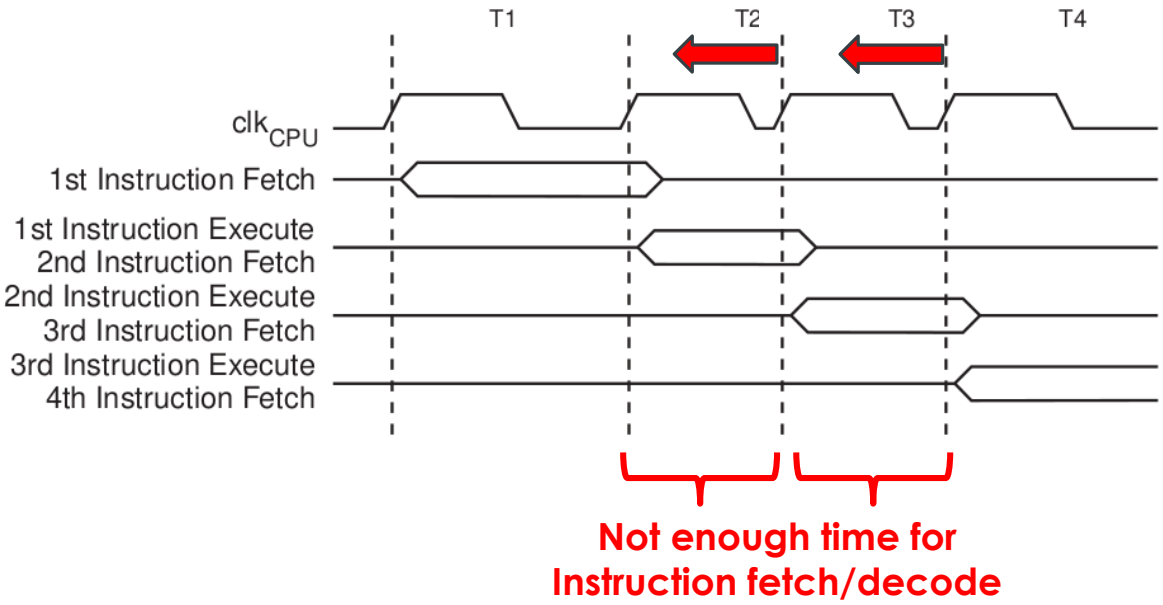
```
login: root
Password:

Login incorrect.

login: root
Password:

Login incorrect.

Last login: Wed Dec 31 12:34:56 2025 from 127.0.0.1
root@iotvictim:~#
```

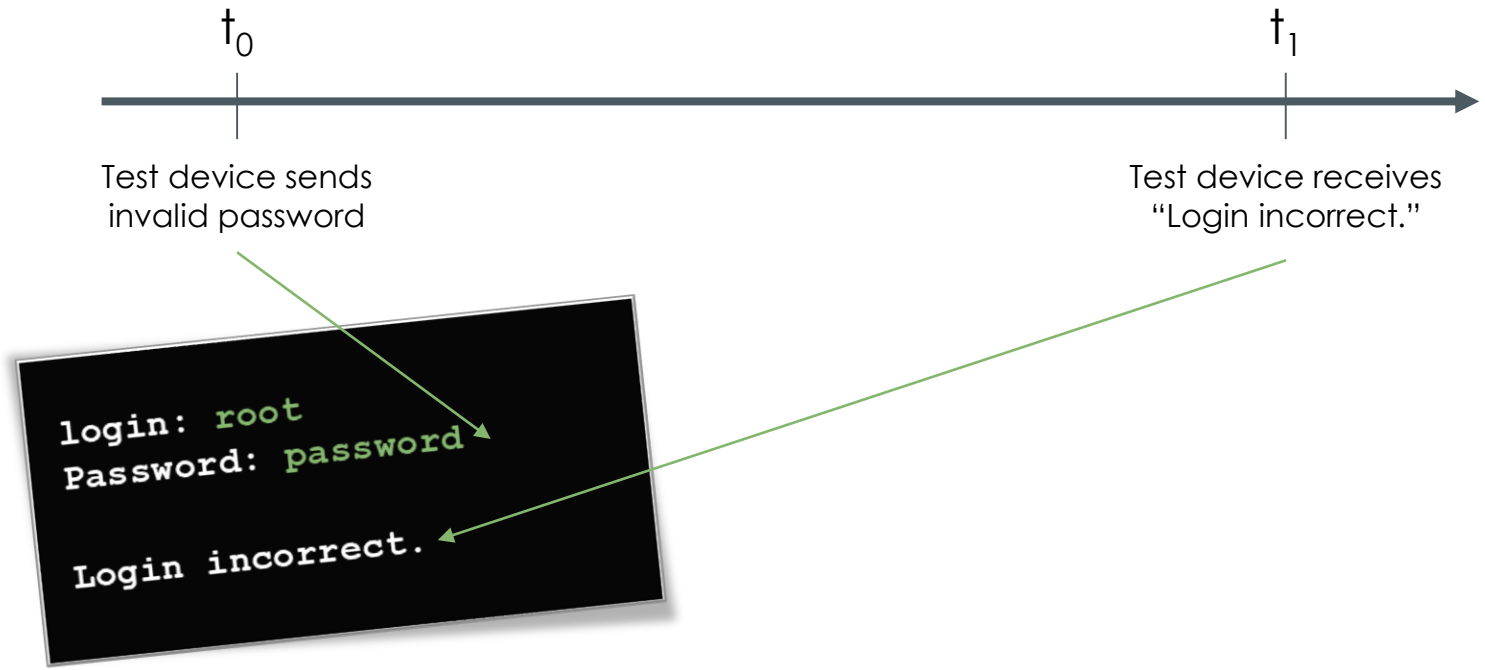

ATMEL 35473D
ATMEGA328P U
19338V4

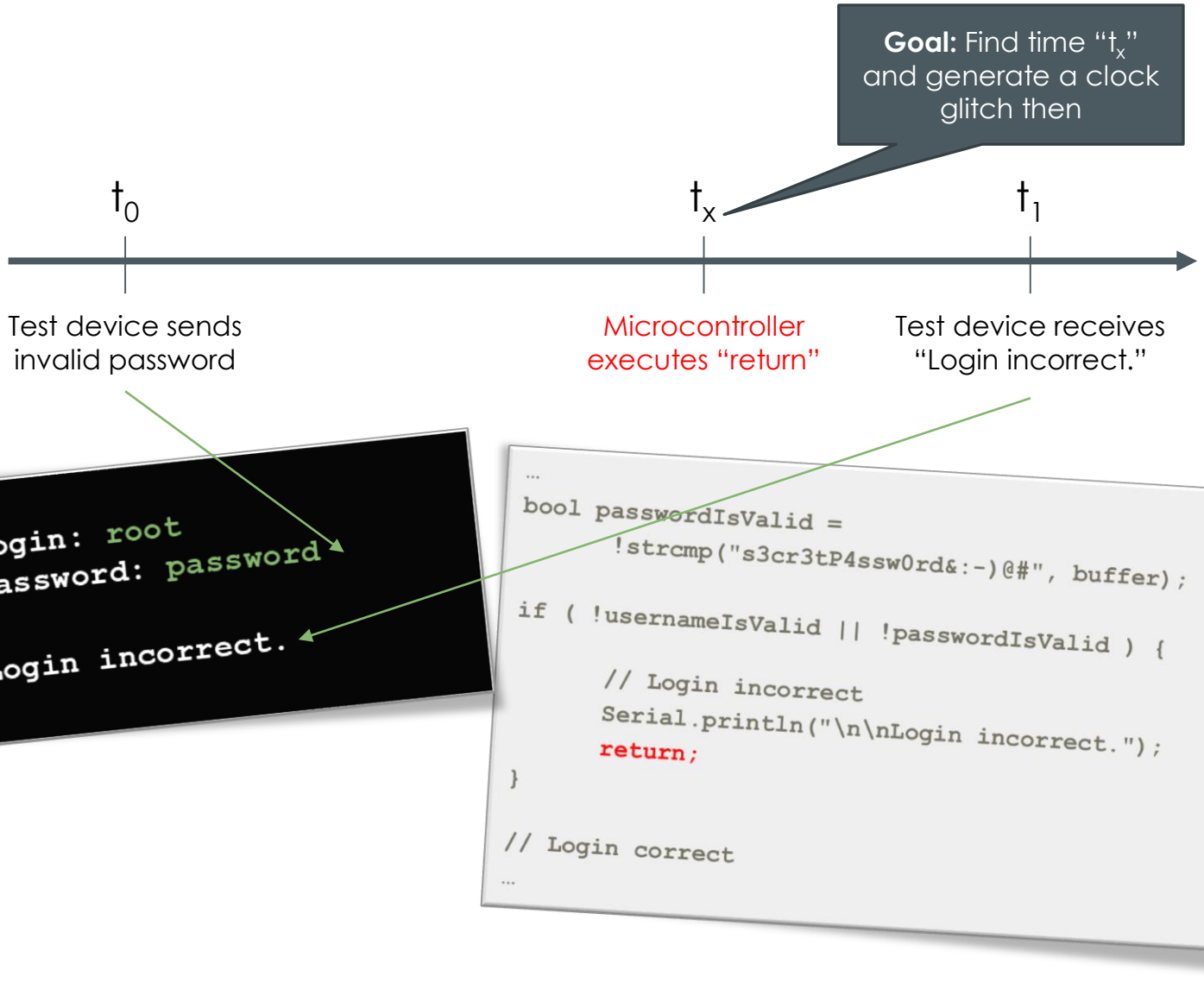

USB Output
Operating Mode
Custom Security Test Tool (ESP32)
• Serial I/O
• Clock Signal
• Power
Device Under Test (ATMega238P)
Status Display
IDLE
3.3V/8MHz/28800-N-8-1

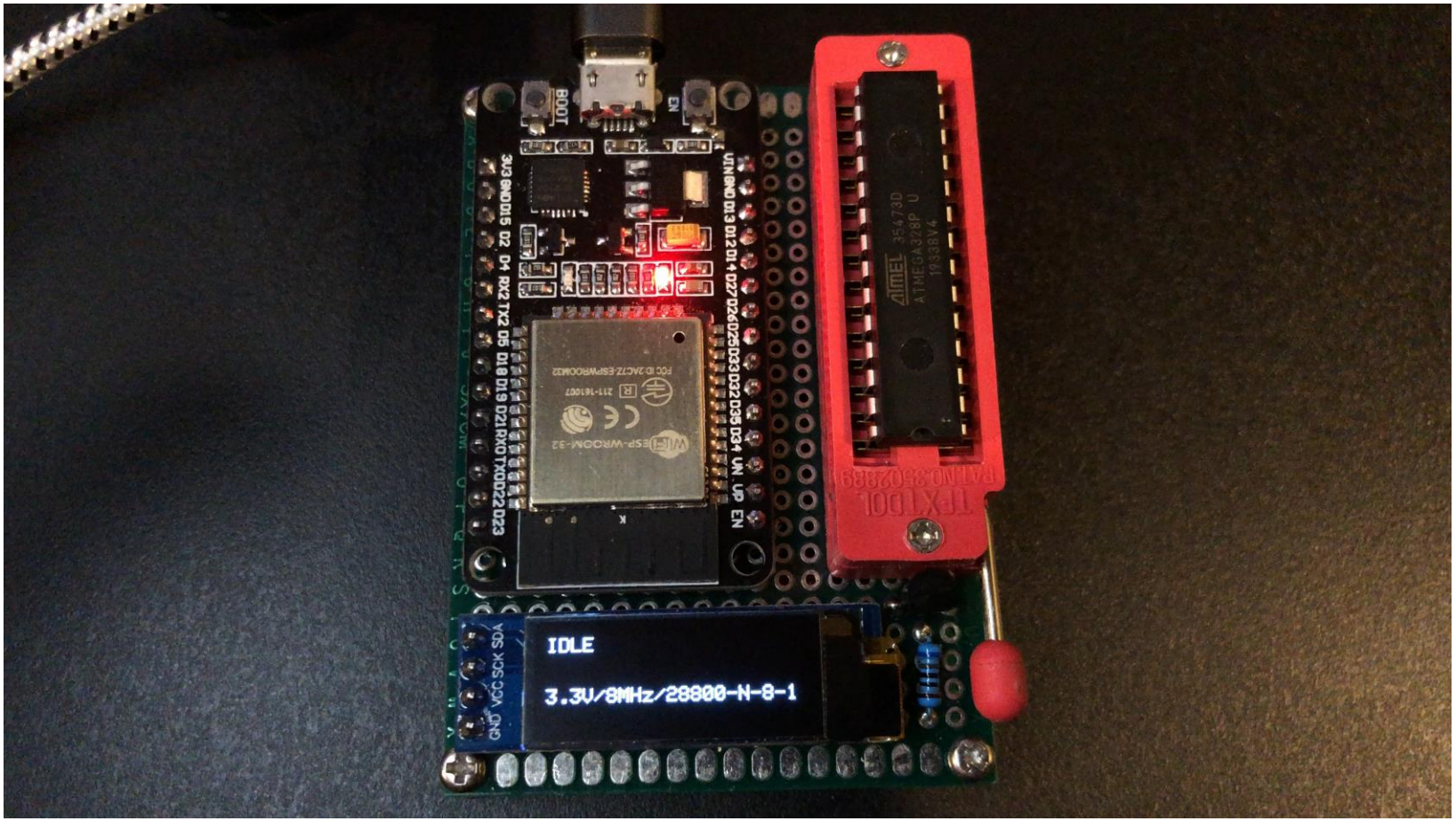
Clock Glitching

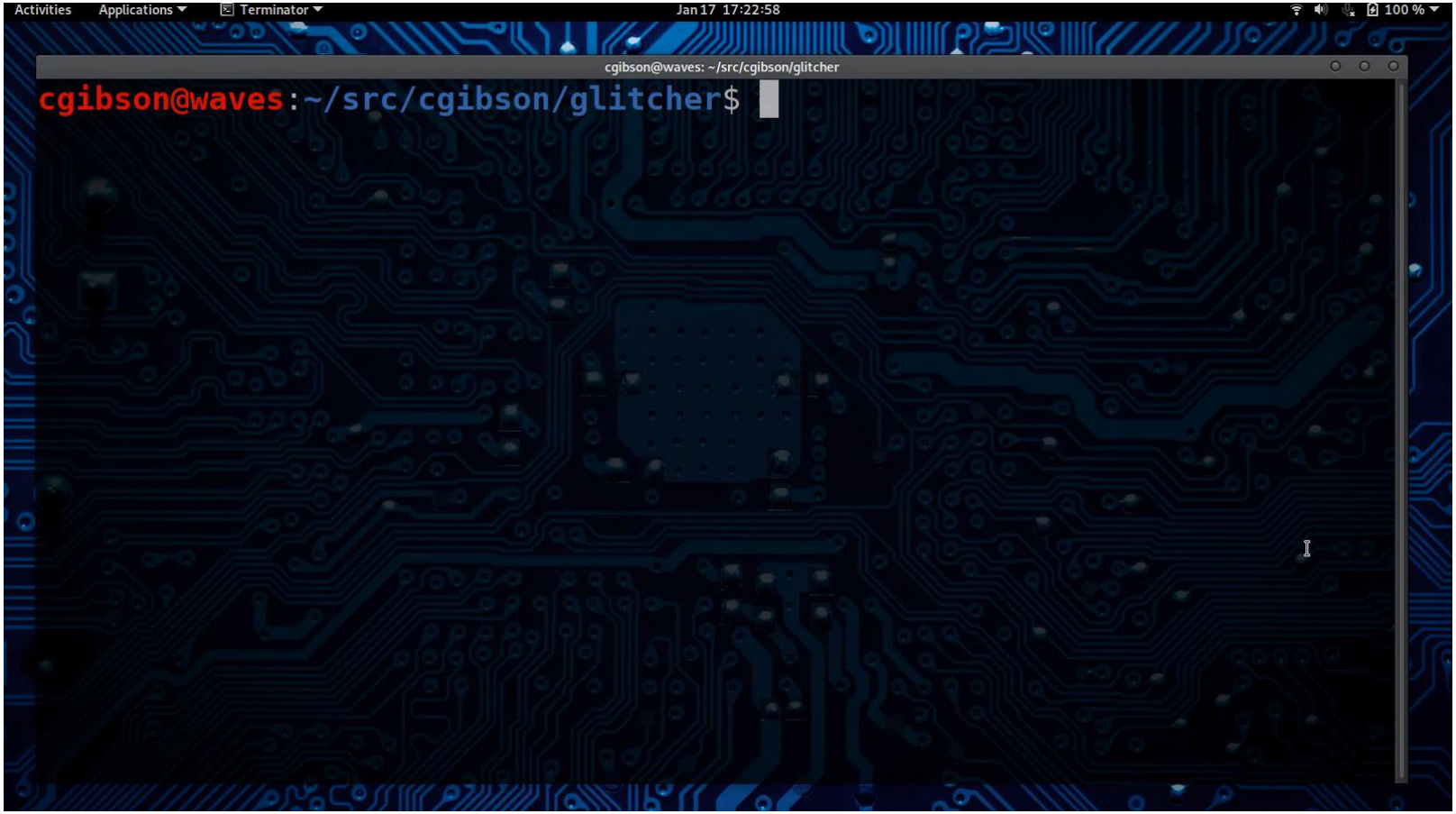
What happens if we introduce a very brief, rapid series of clock pulses? (*e.g.*, 30-40x the normal clock speed?)

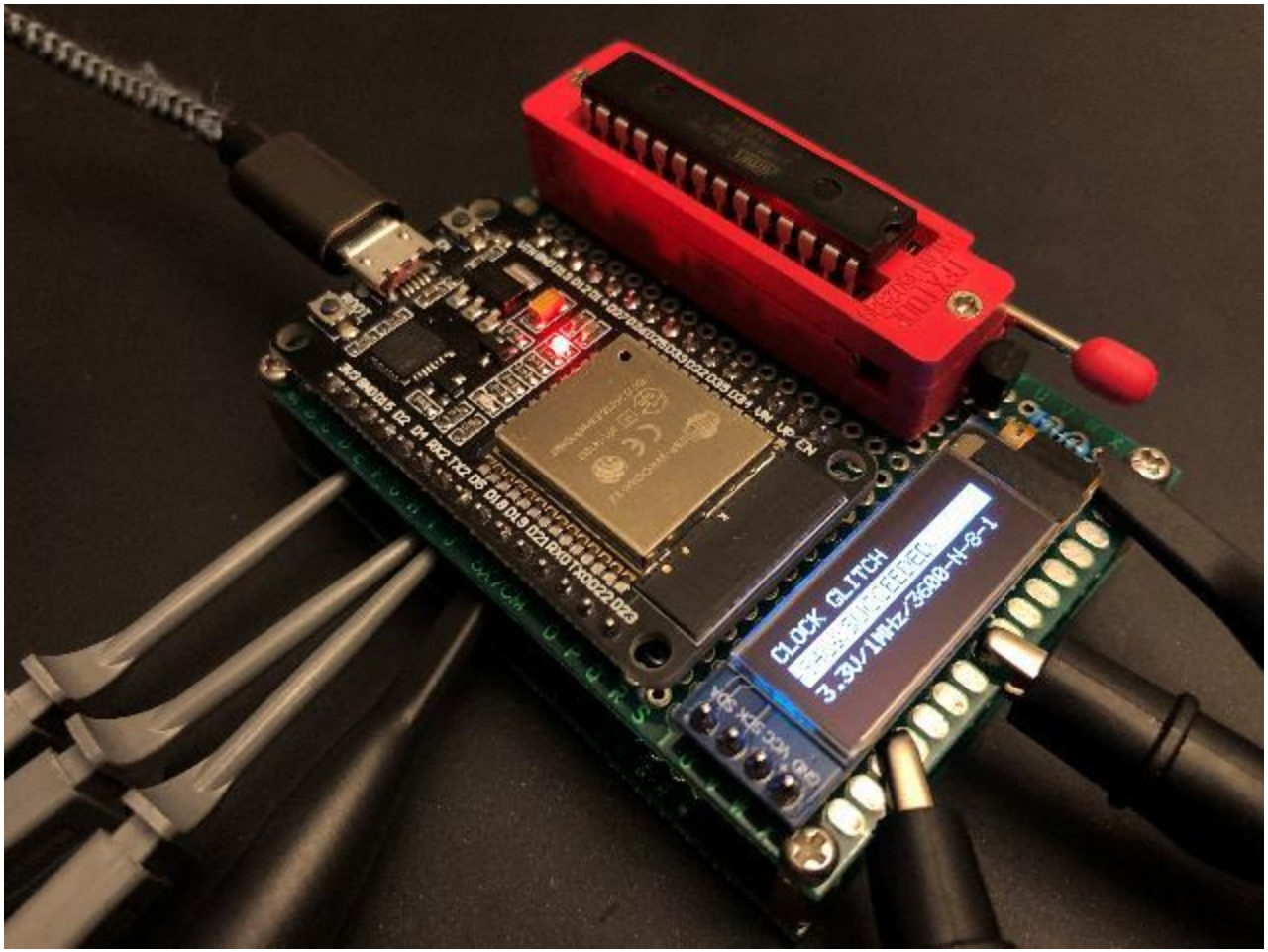


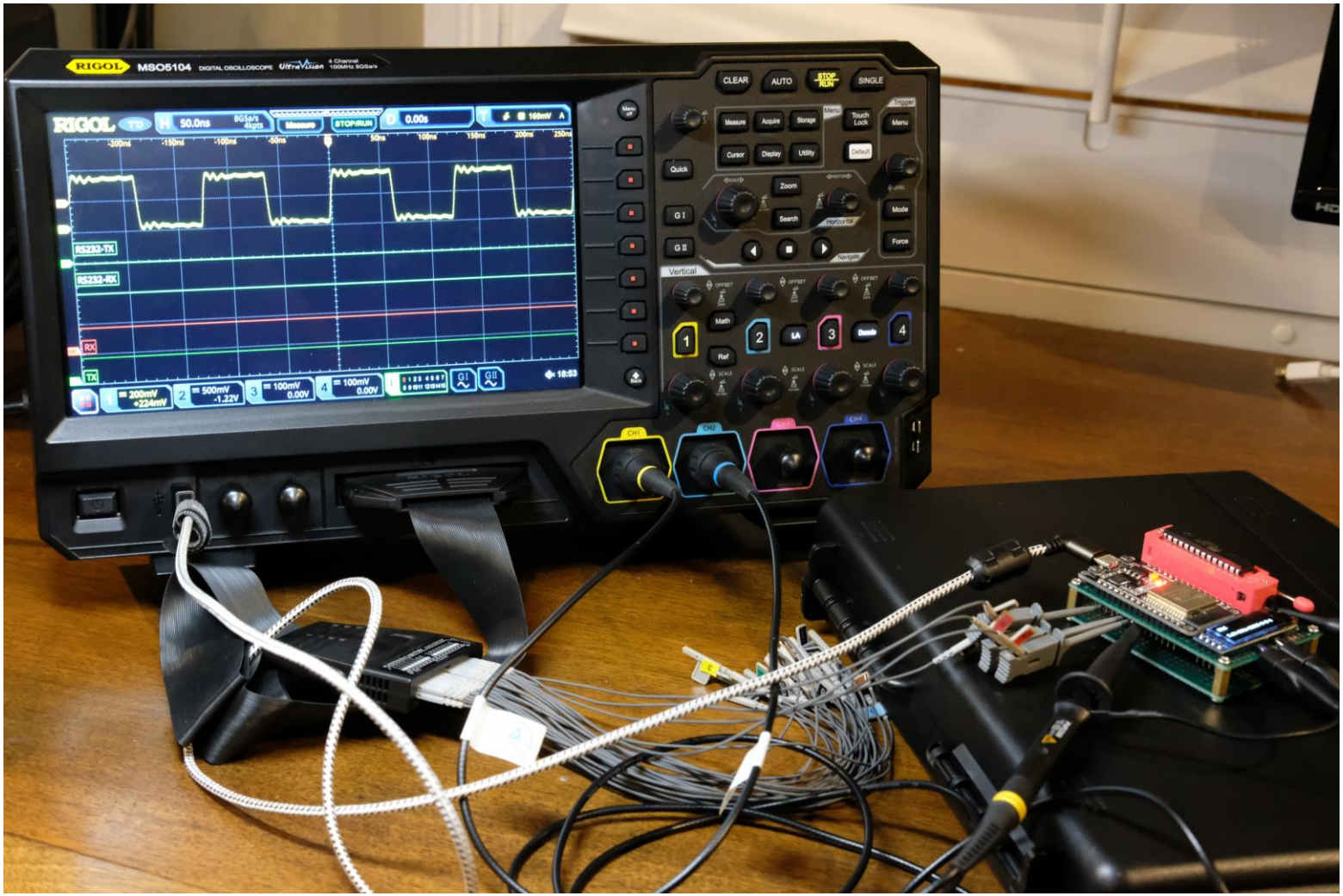

t_0
t_1
Test device sends invalid password
Test device receives "Login incorrect."
login: root
Password: password
Login incorrect.


Goal: Find time "t_x" and generate a clock glitch then
t_0
t_x
t_1
Test device sends invalid password
Microcontroller executes "return"
Test device receives "Login incorrect."
login: root
Password: password
Login incorrect.
…
bool passwordIsValid =
!strcmp("s3cr3tP4ssw0rd&:-)@#", buffer);
if (!usernameIsValid || !passwordIsValid) {
// Login incorrect
Serial.println("\n\nLogin incorrect.");
return;
}
// Login correct
…

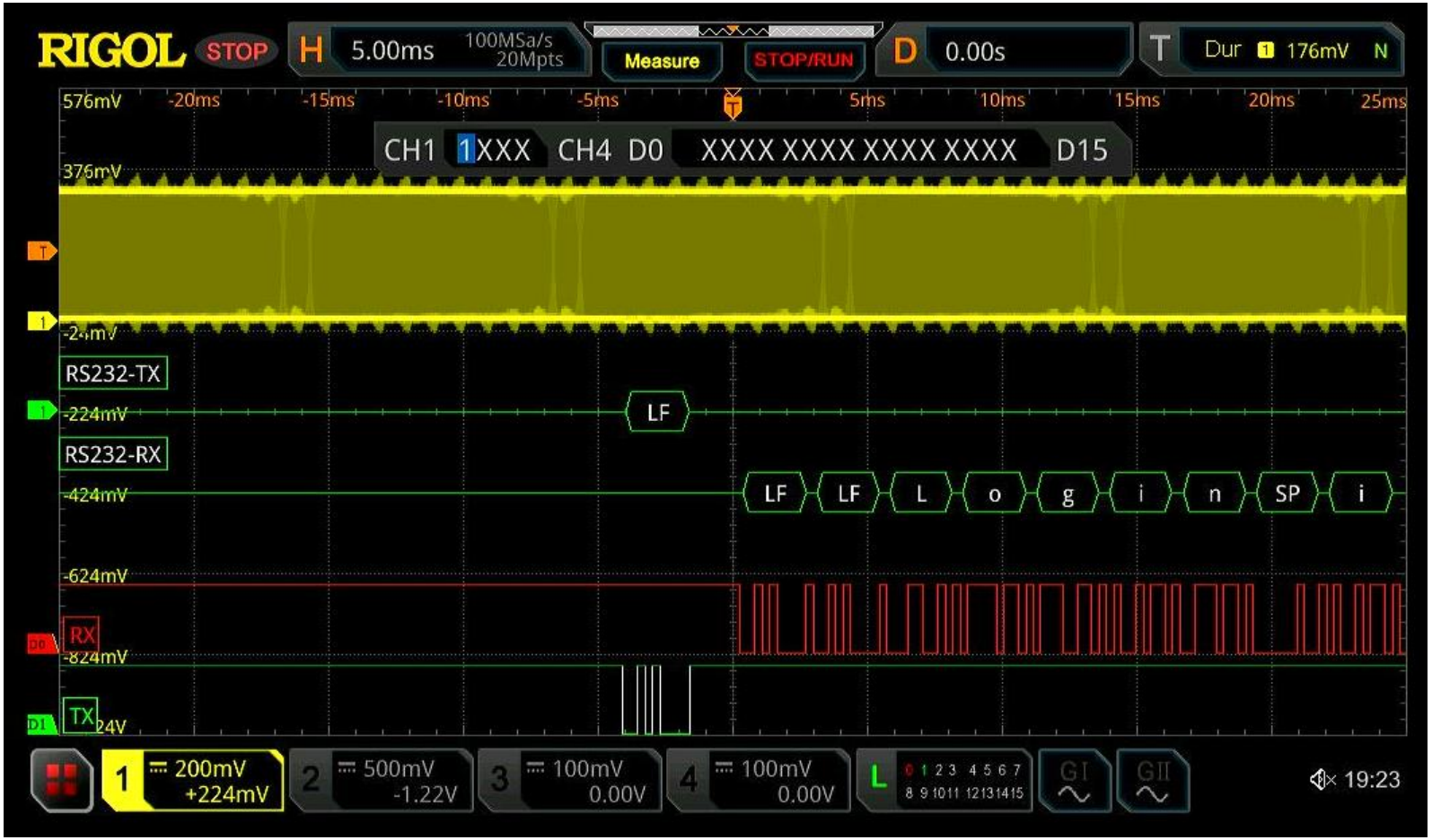

IDLE
3.3V/8MHz/28800-N-8-1
ATMEGA328P
ESP-WROOM-32

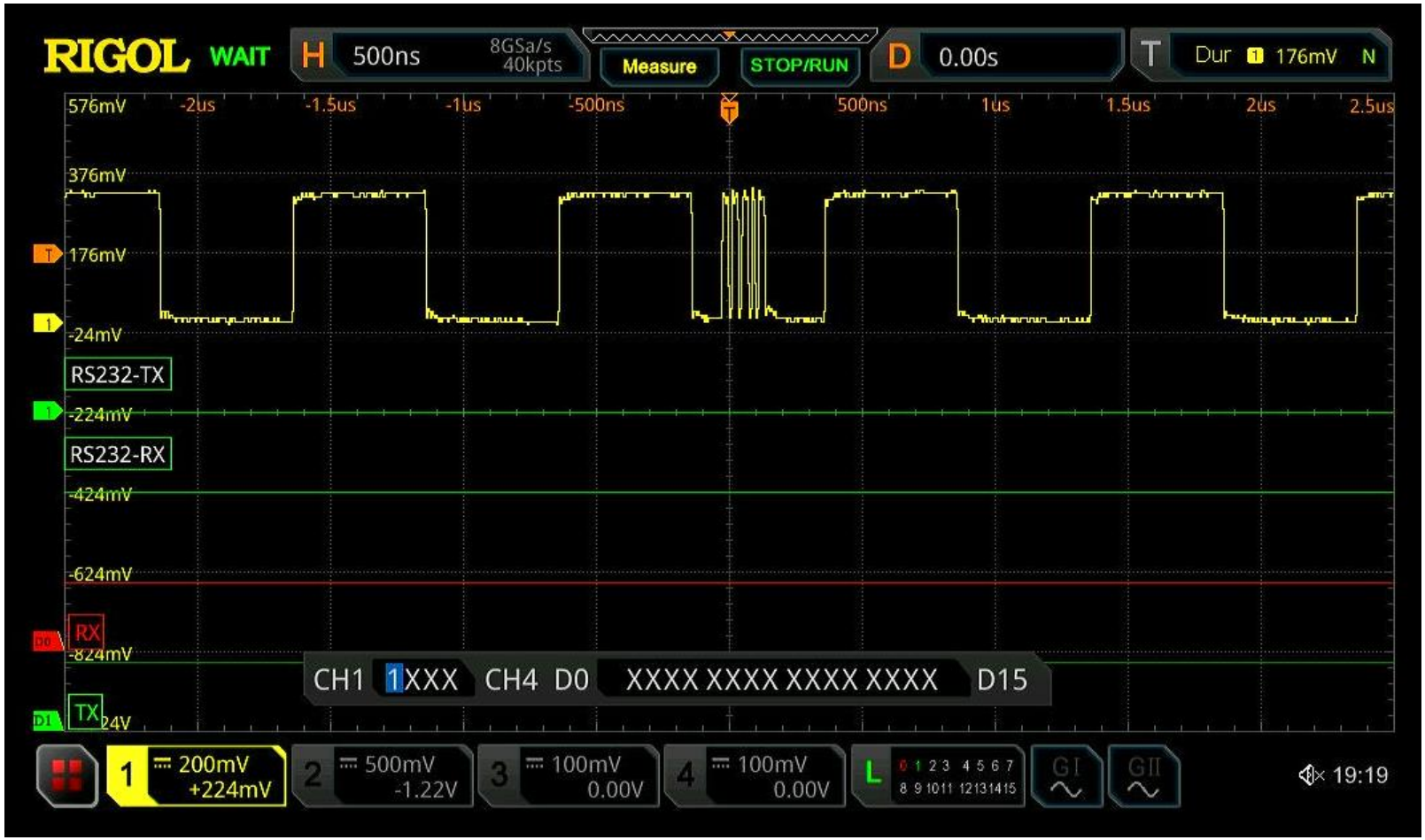

Activities
Applications
Terminator
Jan 17 17:22:58
100 %
cgibson@waves: ~/src/cgibson/glitcher
cgibson@waves:~/src/cgibson/glitcher$

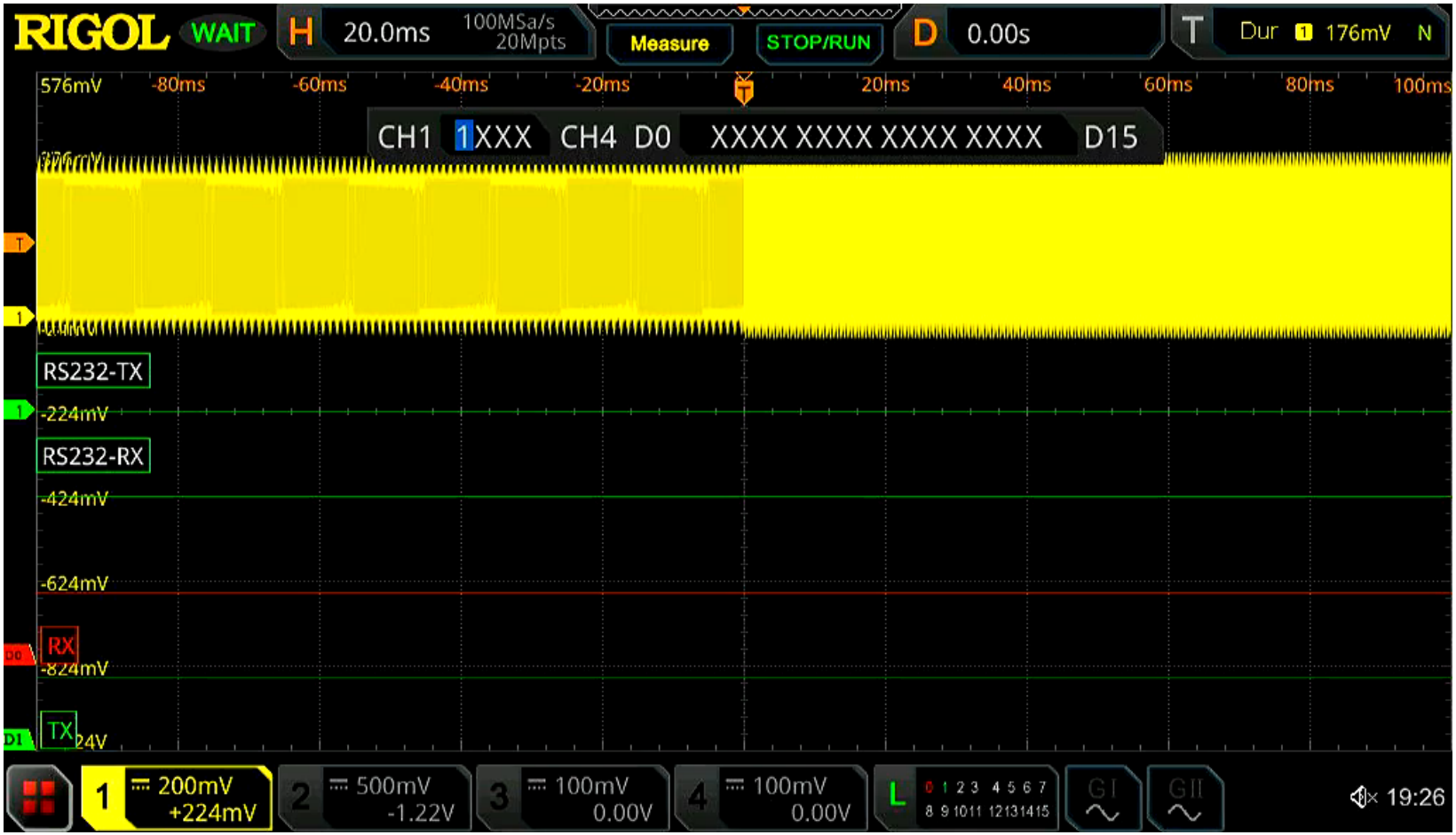

CLOCK GLITCH
3.3V/1MHz/36BB-N-8-1

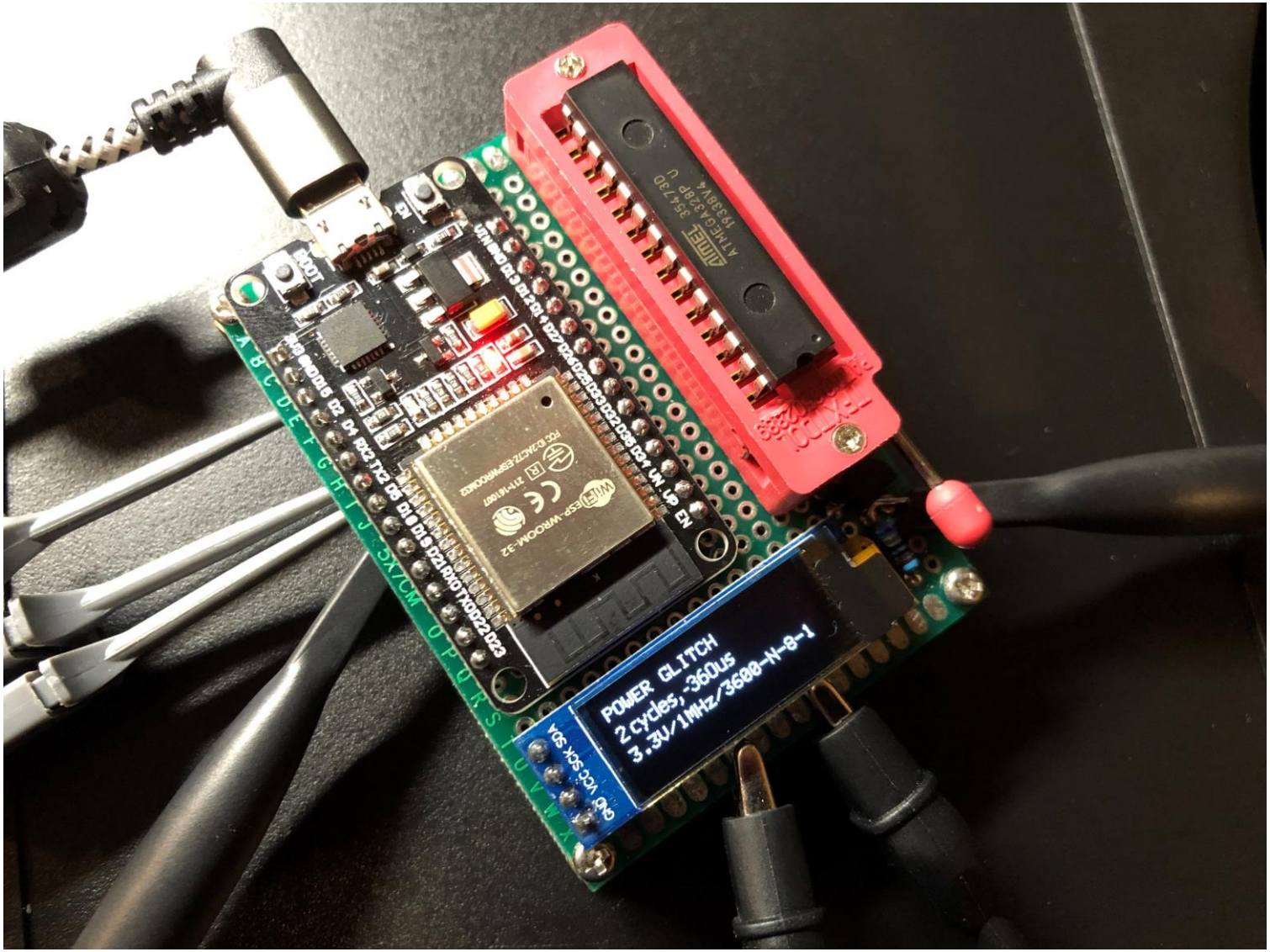

RIGOL
MSO5104
DIGITAL OSCILLOSCOPE
CLEAR
AUTO
STOP RUN
SINGLE
RS232-TX
RS232-RX
RX
TX
Vertical

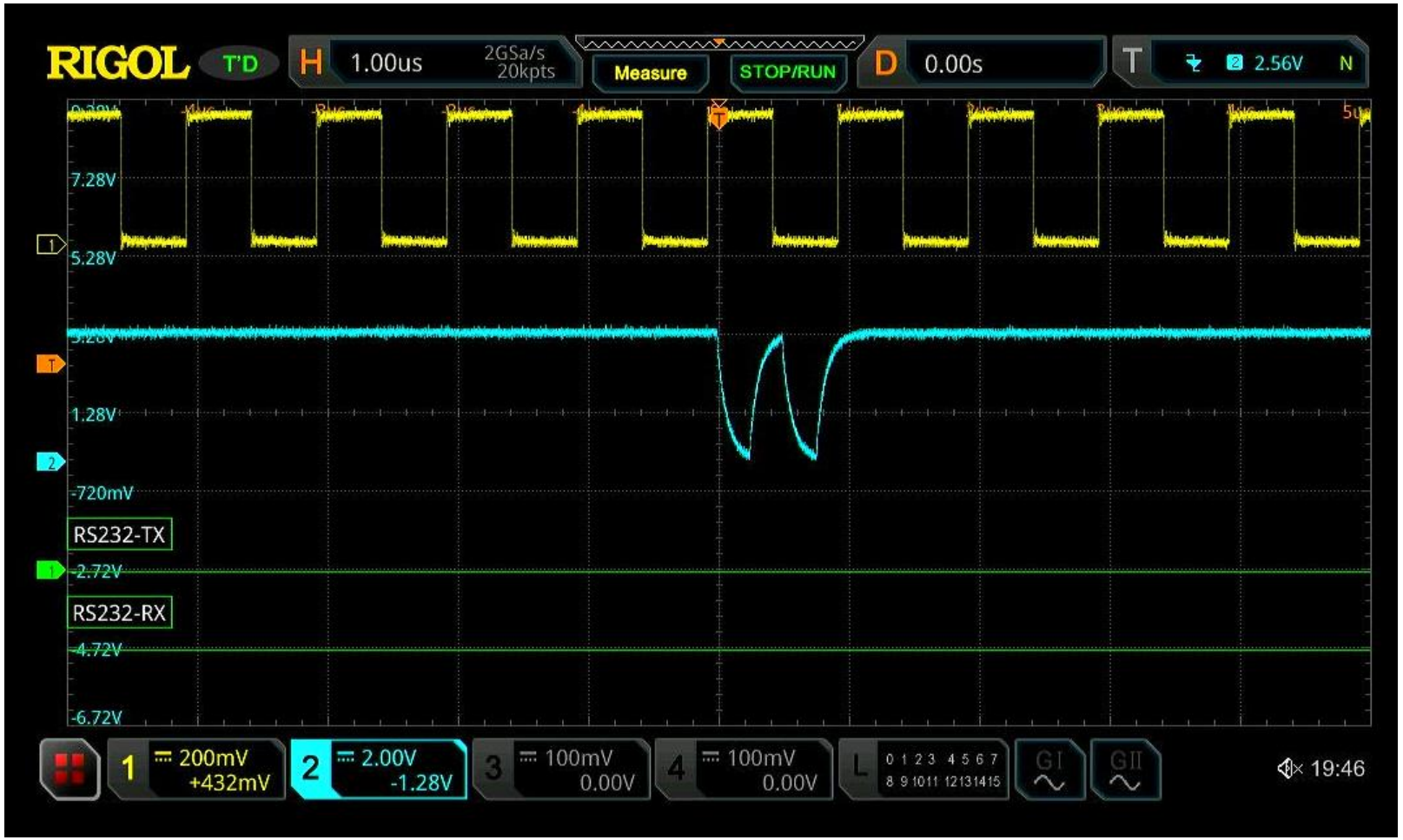

RIGOL
T'D
H
2.00ms
1GSa/s
20Mpts
Measure
STOP/RUN
D
0.00s
T
1
176mV
A
-8ms
-6ms
-4ms
-2ms
2ms
4ms
6ms
8ms
10ms
RX
TX
1
200mV
+224mV
2
500mV
-1.22V
3
100mV
0.00V
4
100mV
0.00V
L
GI
GII
19:06

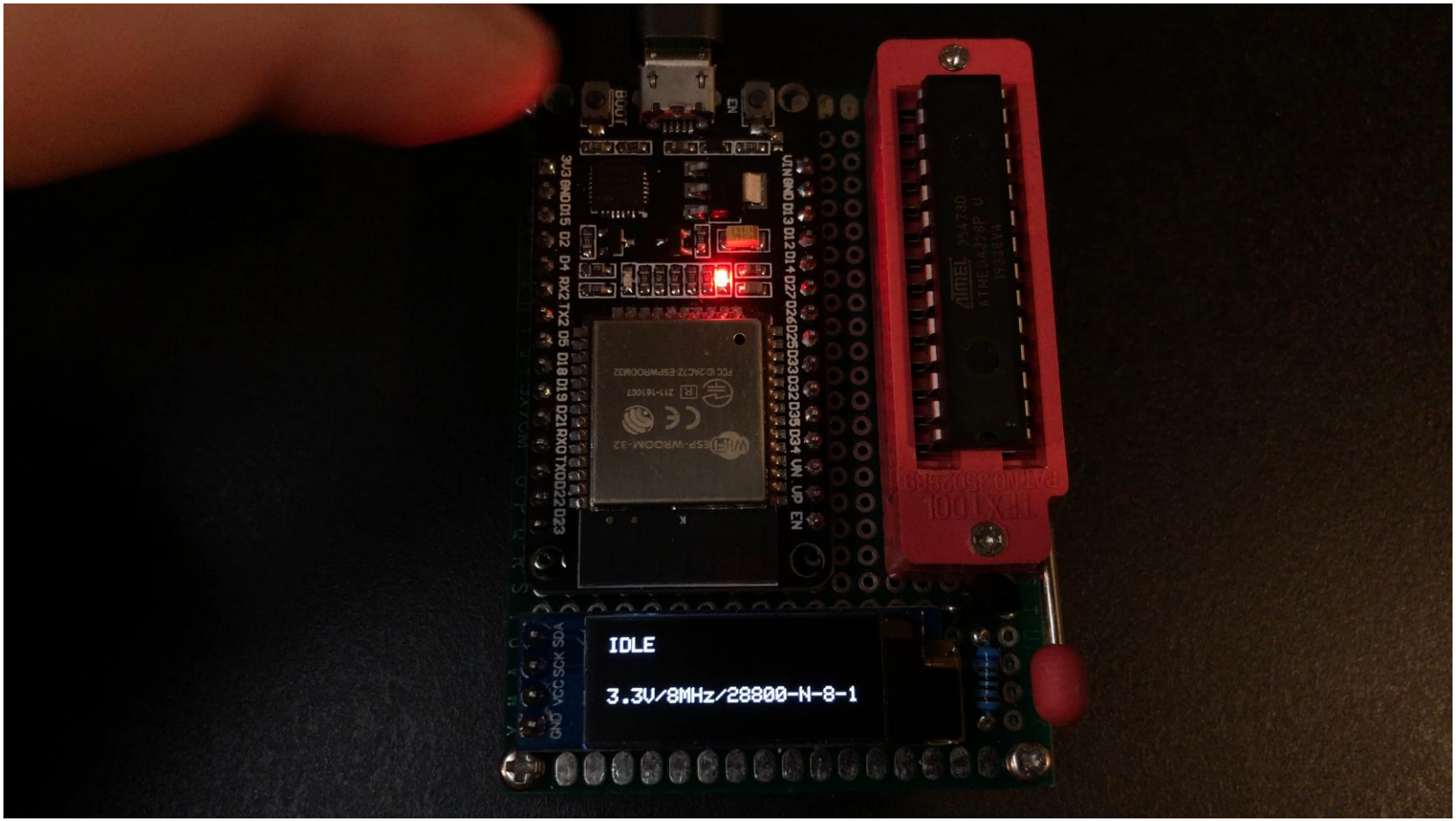

RIGOL
STOP
H 5.00ms 100MSa/s 20Mpts
Measure
STOP/RUN
D 0.00s
T Dur 1 176mV N
576mV -20ms -15ms -10ms -5ms 5ms 10ms 15ms 20ms 25ms
CH1 1XXX CH4 D0 XXXX XXXX XXXX XXXX D15
376mV
-24mV
RS232-TX
-224mV
LF
RS232-RX
-424mV
LF LF L o g i n SP i
-624mV
RX
-824mV
TX
24V
1 200mV +224mV
2 500mV -1.22V
3 100mV 0.00V
4 100mV 0.00V
L 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
GI
GII
19:23

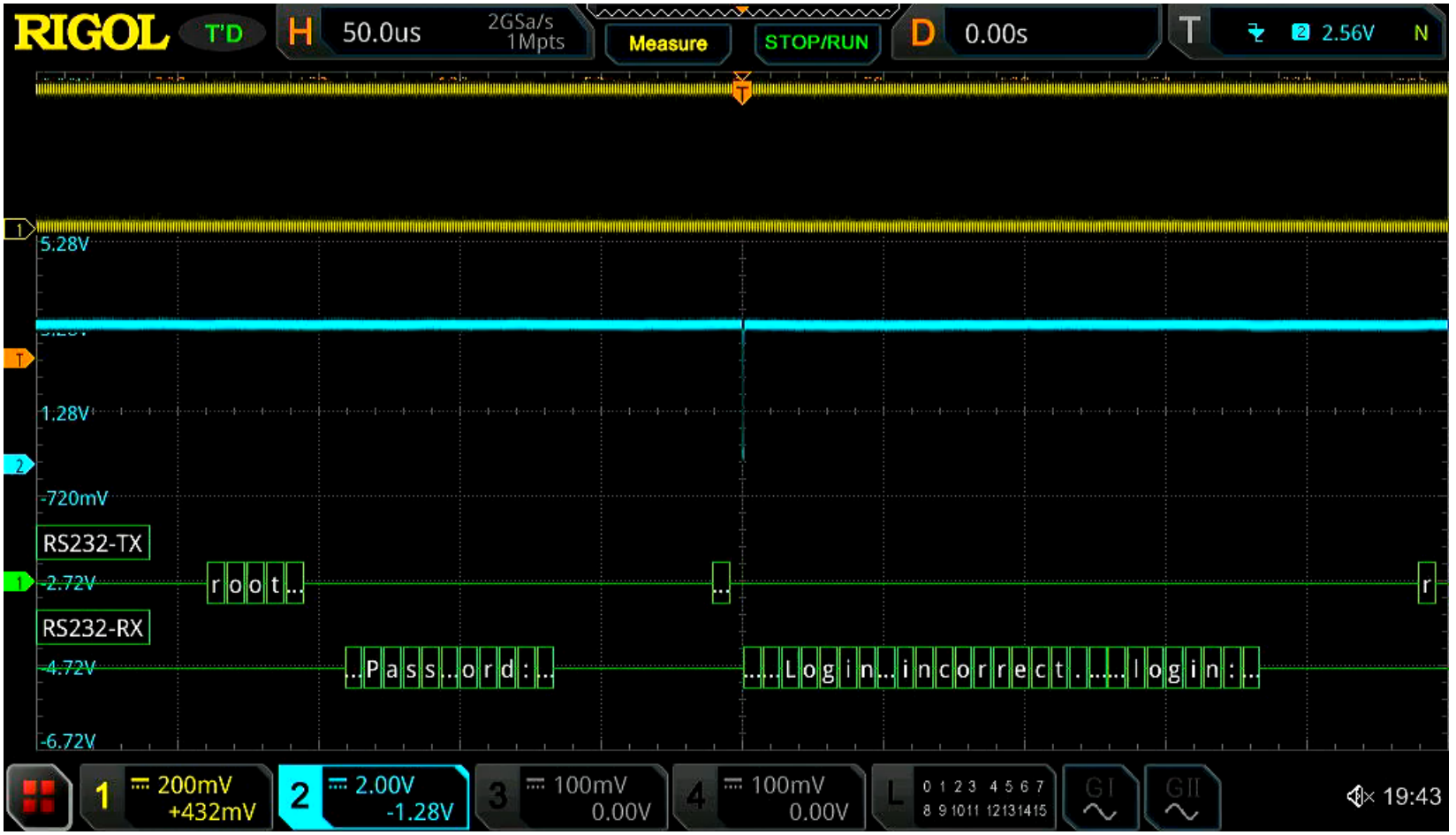

RIGOL
WAIT
H 500ns 8GSa/s 40kpts
Measure
STOP/RUN
D 0.00s
T Dur 1 176mV N
576mV -2us -1.5us -1us -500ns 500ns 1us 1.5us 2us 2.5us
376mV
176mV
-24mV
RS232-TX
-224mV
RS232-RX
-424mV
-624mV
RX
-824mV
TX
CH1 1XXX CH4 D0 XXXX XXXX XXXX XXXX D15
1 200mV +224mV
2 500mV -1.22V
3 100mV 0.00V
4 100mV 0.00V
L
GI
GII
19:19

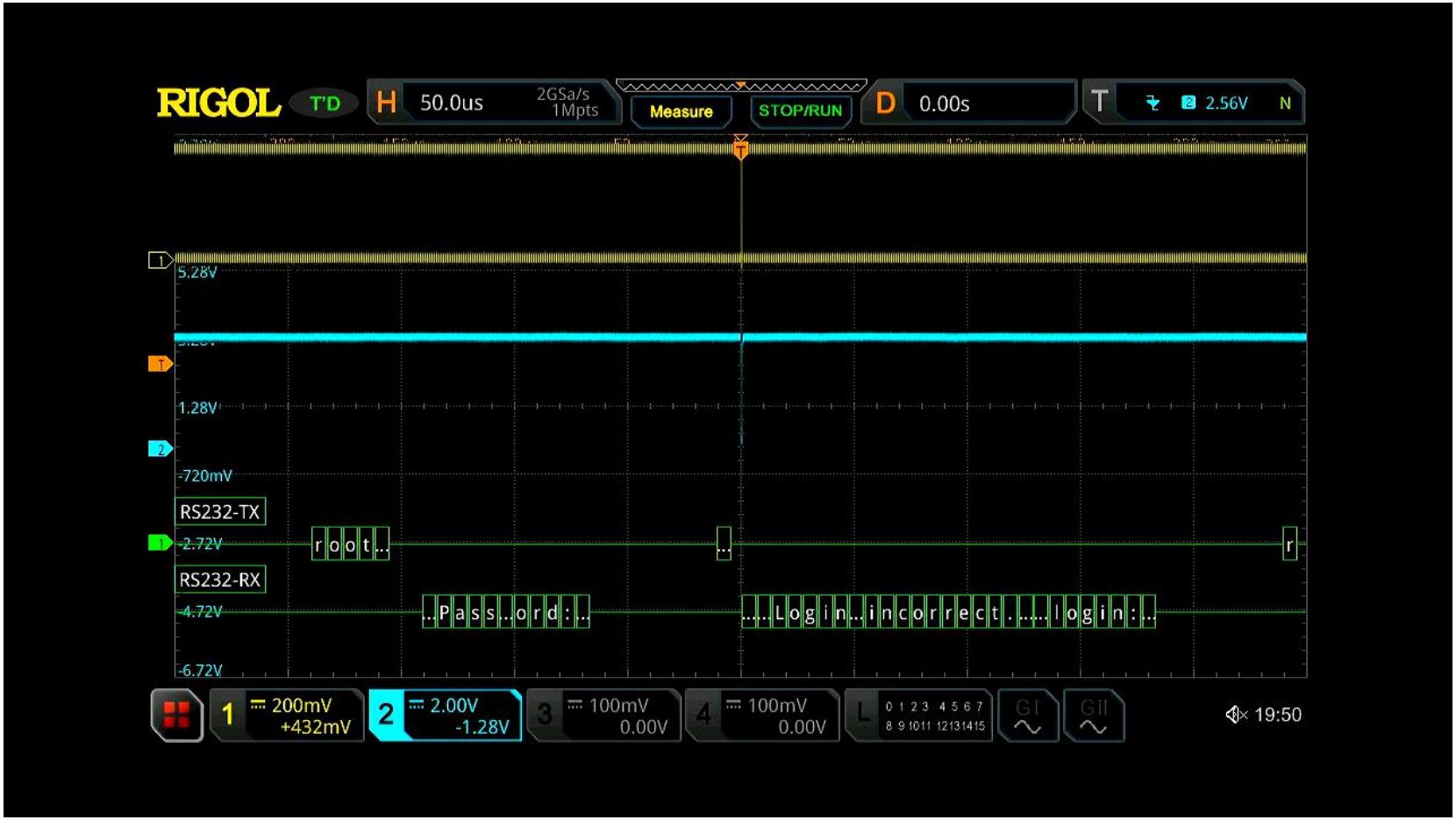

RIGOL
WAIT
H 20.0ms
100MSa/s
20Mpts
Measure
STOP/RUN
D 0.00s
T Dur 1 176mV N
576mV
-80ms
-60ms
-40ms
-20ms
20ms
40ms
60ms
80ms
100ms
CH1 1XXX CH4 D0 XXXX XXXX XXXX XXXX D15
RS232-TX
-224mV
RS232-RX
-424mV
-624mV
RX
TX
1 200mV +224mV
2 500mV -1.22V
3 100mV 0.00V
4 100mV 0.00V
19:26


POWER GLITCH
2 cycles, -360us
3.3V/1MHz/3600-N-8-1
ESP-WROOM-32
ATMEGA328P


RIGOL
T'D
H 1.00us 2GSa/s 20kpts
Measure
STOP/RUN
D 0.00s
T 2.56V N
7.28V
5.28V
1.28V
-720mV
RS232-TX
-2.72V
RS232-RX
-4.72V
-6.72V
1 200mV +432mV
2 2.00V -1.28V
3 100mV 0.00V
4 100mV 0.00V
19:46


IDLE
3.3V/8MHz/28800-N-8-1

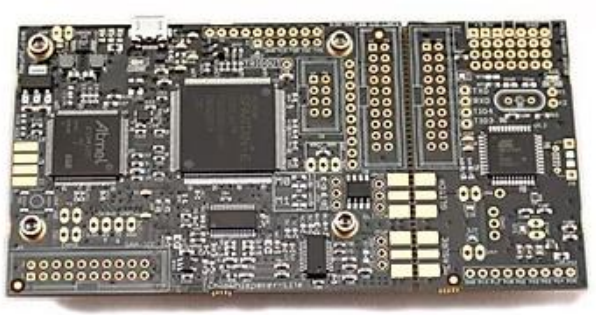

RIGOL
T'D
H 50.0us 2GSa/s 1Mpts
Measure
STOP/RUN
D 0.00s
T 2.56V N
5.28V
1.28V
-720mV
RS232-TX
-2.72V
root..
..
r
RS232-RX
-4.72V
..Pass..ord:..
.....Login..incorrect.....login:..
-6.72V
1 200mV +432mV
2 2.00V -1.28V
3 100mV 0.00V
4 100mV 0.00V
0 1 2 3 4 5 6 7
8 9 1011 12131415
19:43


RIGOL
T'D
H 50.0us 2GSa/s 1Mpts
Measure
STOP/RUN
D 0.00s
T 2.56V N
5.28V
1.28V
-720mV
RS232-TX
-2.72V
root..
..
r
RS232-RX
-4.72V
..Password:..
...Login..incorrect.....login:..
-6.72V
1 200mV +432mV
2 2.00V -1.28V
3 100mV 0.00V
4 100mV 0.00V
19:50

```
login: root
Password:

Login incorrect.

login: root
Password:

Login incorrect.

Last login: Wed Dec 31 12:34:56 2025 from 127.0.0.1
root@iotvictim:~#
```

Fault Injection Attacks

Unlooper
($100USD)



Chip Whisperer
($250USD)

Reference: *Secure Application Programming in the Presence of Side Channel Attacks*, Whitteman et al., RSA Conference 2008
http://www.riscure.com/benzine/documents/Paper_Side_Channel_Patterns.pdf

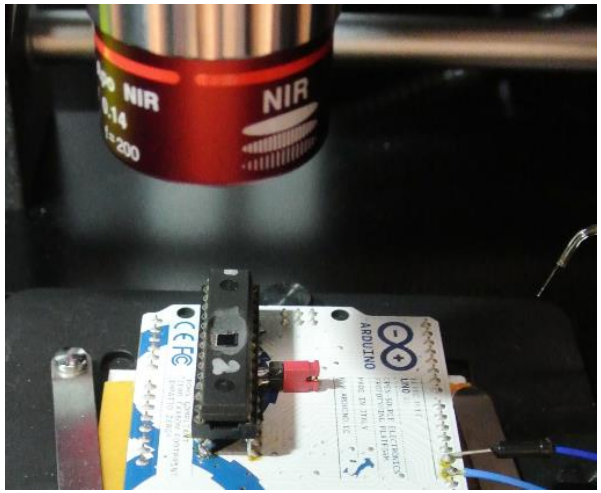
Fault Injection Attacks

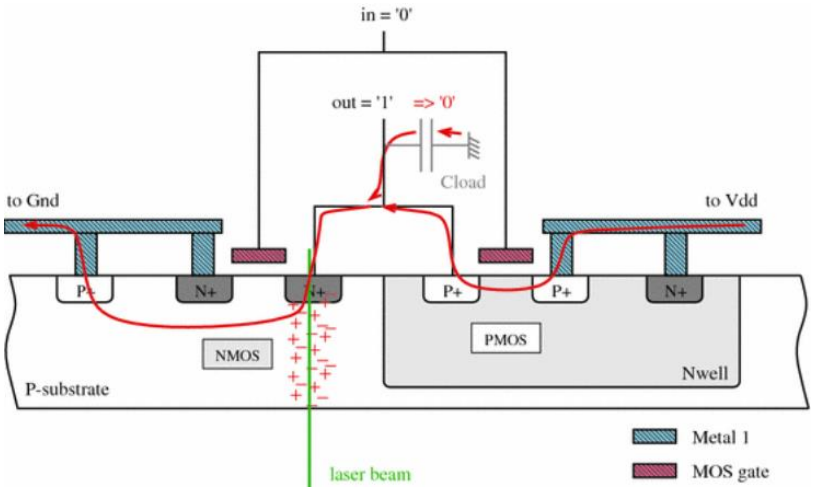
- Various researchers are successfully using ML techniques to find the correct duration/timing.
- Many other ways to cause controlled circuit malfunctions:
  - Laser
  - Strobes
  - Focussed EM pulses

Fault Injection Attacks

Properly-focussed laser or electromagnetic pulses can flip the output bits of logic gates.

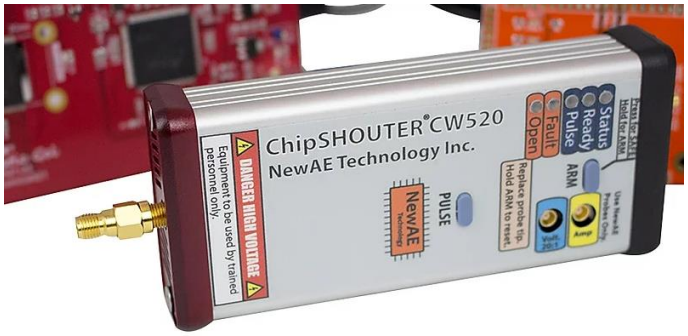
- Automated test tools exist that can automatically scan a chip surface (x-/y-/z-axis)





Source: J. Brier, D. Jap. *Testing Feasibility of Back-Side Laser Fault Injection on a Microcontroller.* (2015)

Fault Injection Attacks



Chip Shouter
(EM Pulse Generation: $3300USD)

Laser Station 2
(Laser Glitch Generation: $$$)

Reverse Engineering

Reverse Engineering

Generally refers to analyzing a product, in order to learn something about its design that its creator wanted to keep *secret*.

- **Hardware:** circuit logic?
- **Software:** source code?

Often, the goal is breaking encryption, bypassing authentication, finding flaws or determining communications protocols.

Reverse Engineering

Legally complicated:

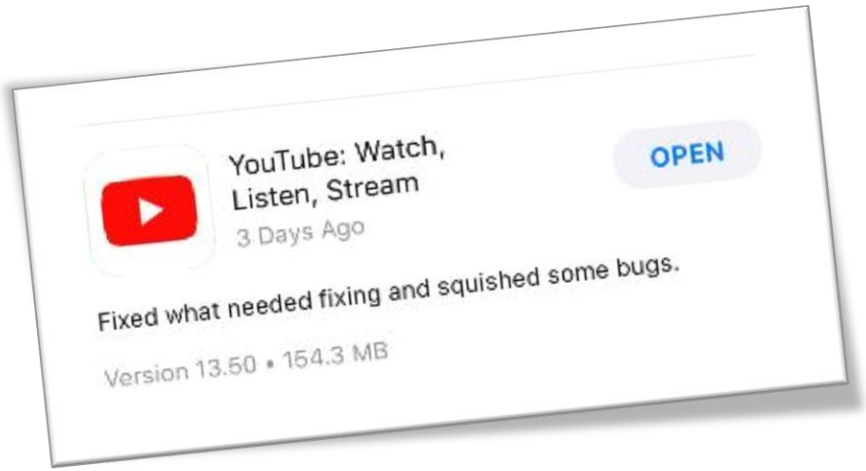
- End-User License Agreements
- Copyright / Trade Secret laws
- US DMCA / EU Directive 2009/24

Often **permitted** for the purposes of achieving "interoperability".

It is usually **forbidden** to circumvent DRM or copy protection.

EULAs may or may not override legislative rights.

Reverse Engineering



PURPOSE. IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY LOSS OF PROFITS, REVENUE, DATA OR DATA USE ARISING FROM THE USE OF THE SOFTWARE OR PROGRAM.

- **Limitations on Reverse Engineering, Decompilation and Disassembly**. You may not reverse engineer (unless required by law for interoperability), disassemble or decompile the SOFTWARE (the forgoing prohibition includes but is not limited to review of data structures or similar materials produced by programs).
- **SOFTWARE as a Component of the PROGRAM –Transfer/Giving/Assigning**. The SOFTWARE is licensed with the PROGRAM as a single integrated product and may only be used with the PROGRAM. If the SOFTWARE is not

Reverse Engineering

Software security application: hackers frequently use reverse-engineering tools to:

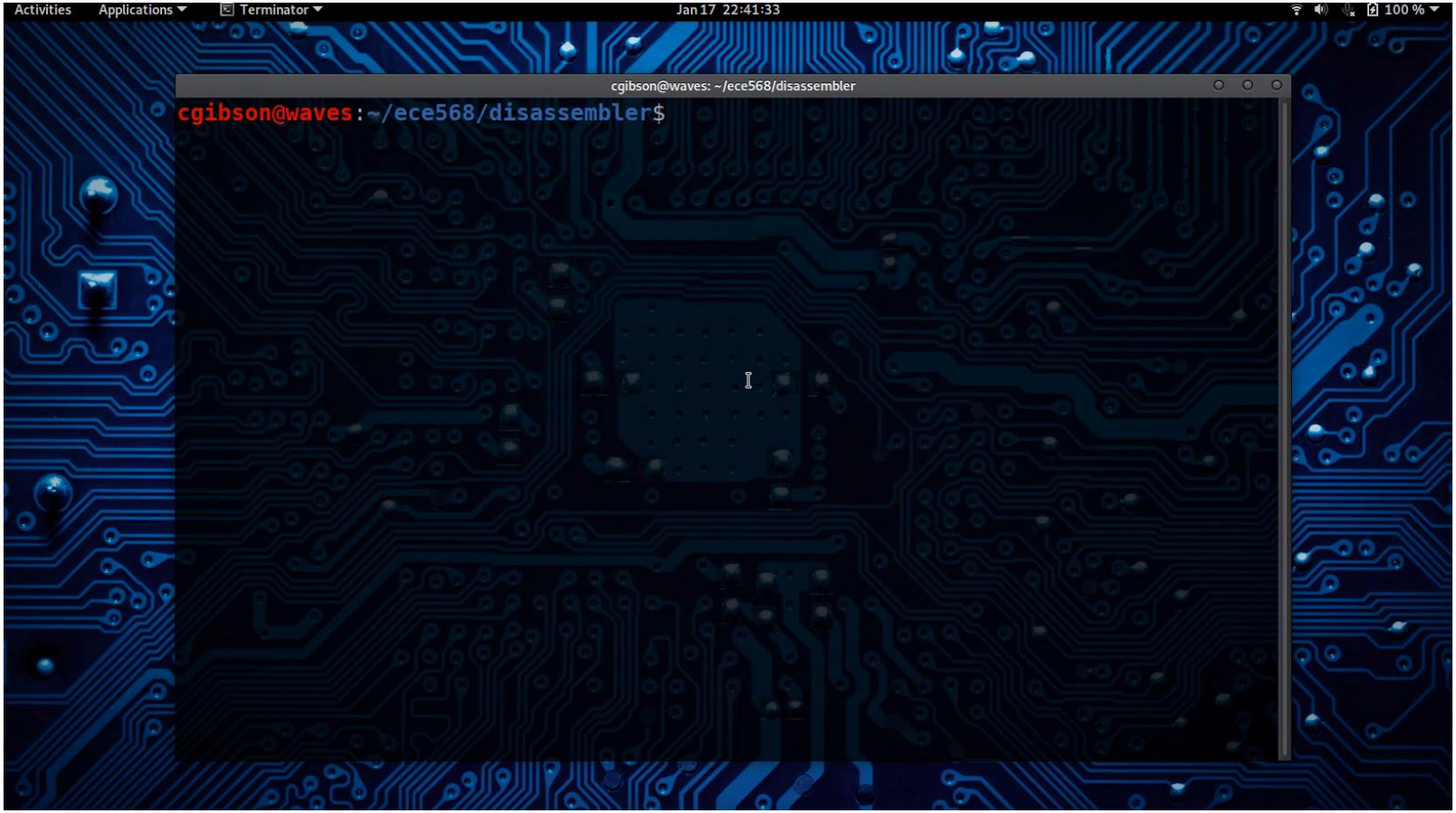
- Uncover flaws in applications
- Analyze security updates, to find what developers have fixed
- Bypass authentication / authorization

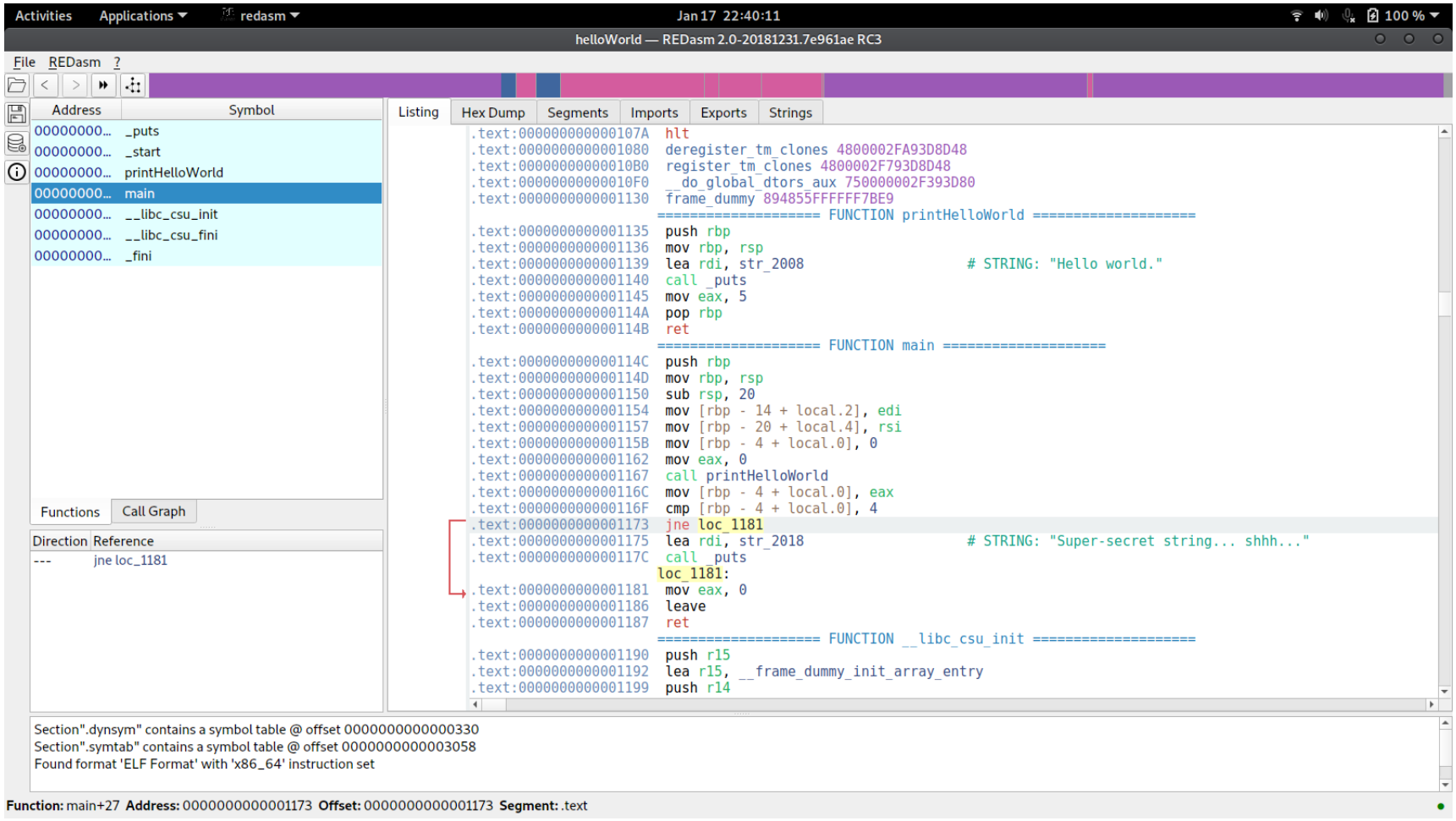
Demo

Reverse-Engineering

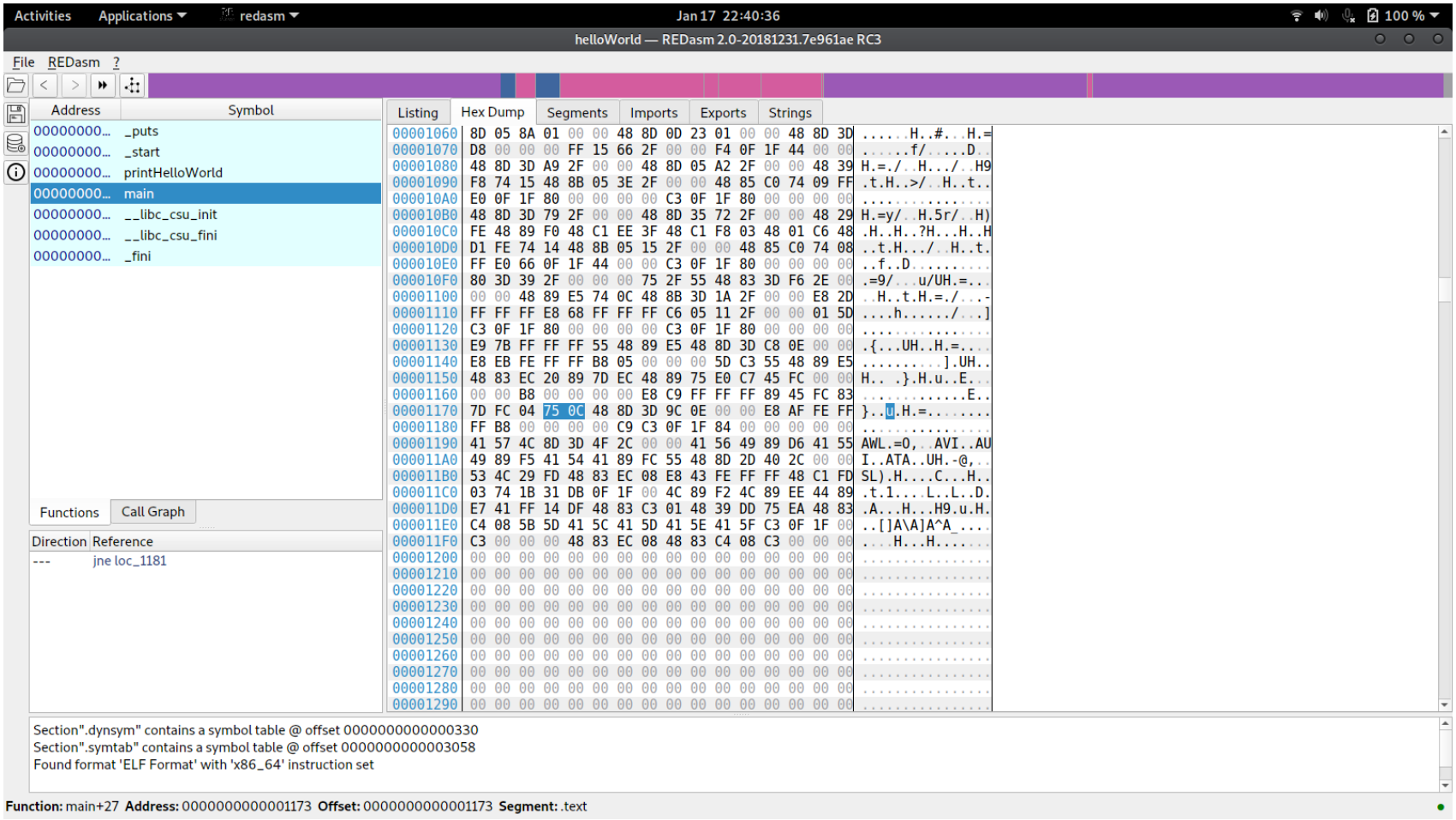
```c
#include <stdio.h>

unsigned int printHelloWorld()
{
        printf("Hello world.\n");
        return(5);
}

int main(int argc, char * argv[])
{
        unsigned int result = 0;
        result = printHelloWorld();
        if ( result == 4 ) {
                printf("Super-secret string... shhh...\n");
        }
        return(0);
}
```

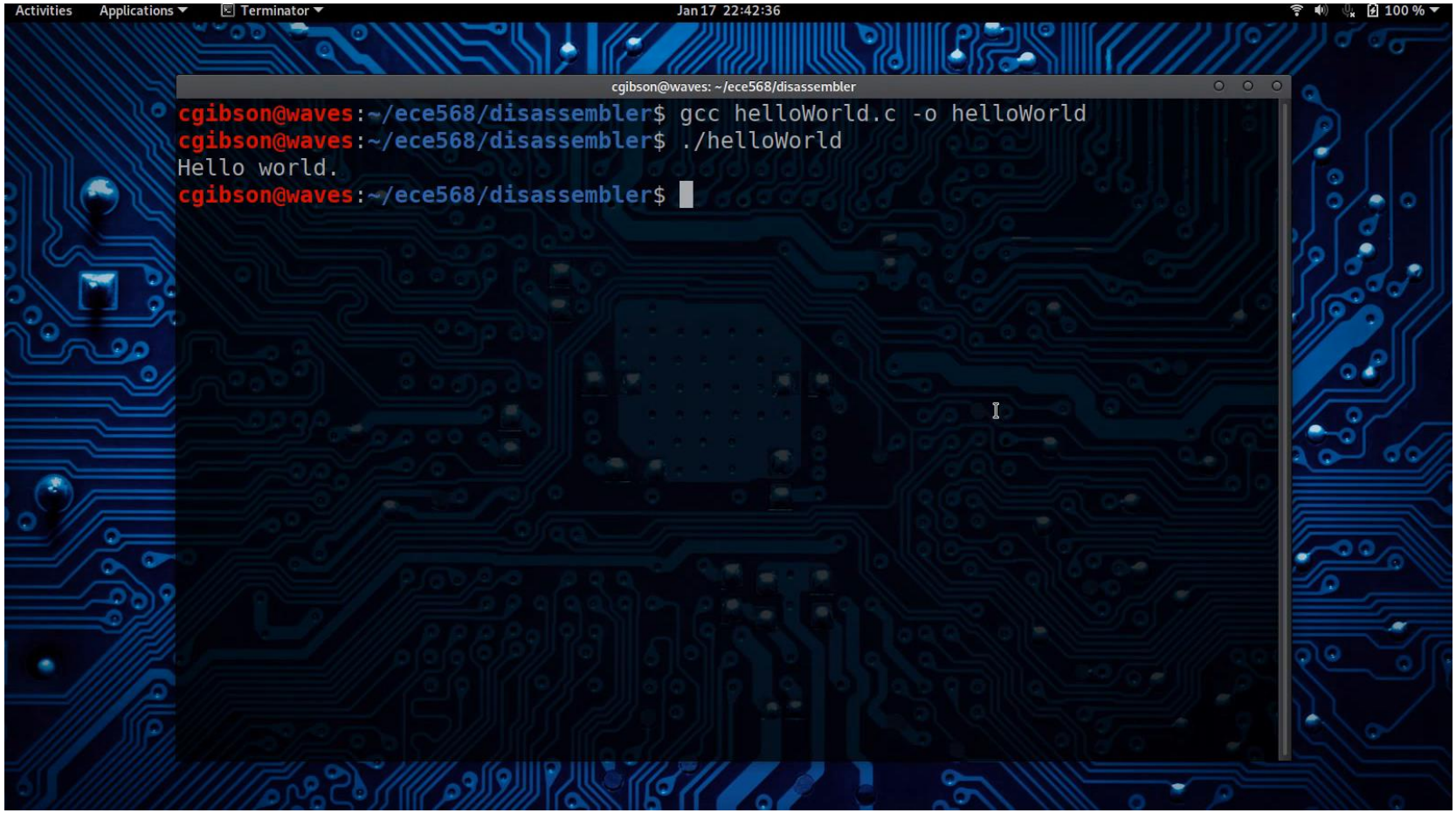
```
#include <stdio.h>

unsigned int printHelloWorld()
{
        printf("Hello world.\n");
        return(5);
}

int main(int argc, char * argv[])
{
        unsigned int result = 0;
        result = printHelloWorld();
        if(result == 4) {
                printf("Super-secret string... shhh...\n");
        }
        return(0);
}
```


Activities
Applications
Terminator
Jan 17 22:41:33
100 %
cgibson@waves: ~/ece568/disassembler
cgibson@waves:~/ece568/disassembler$

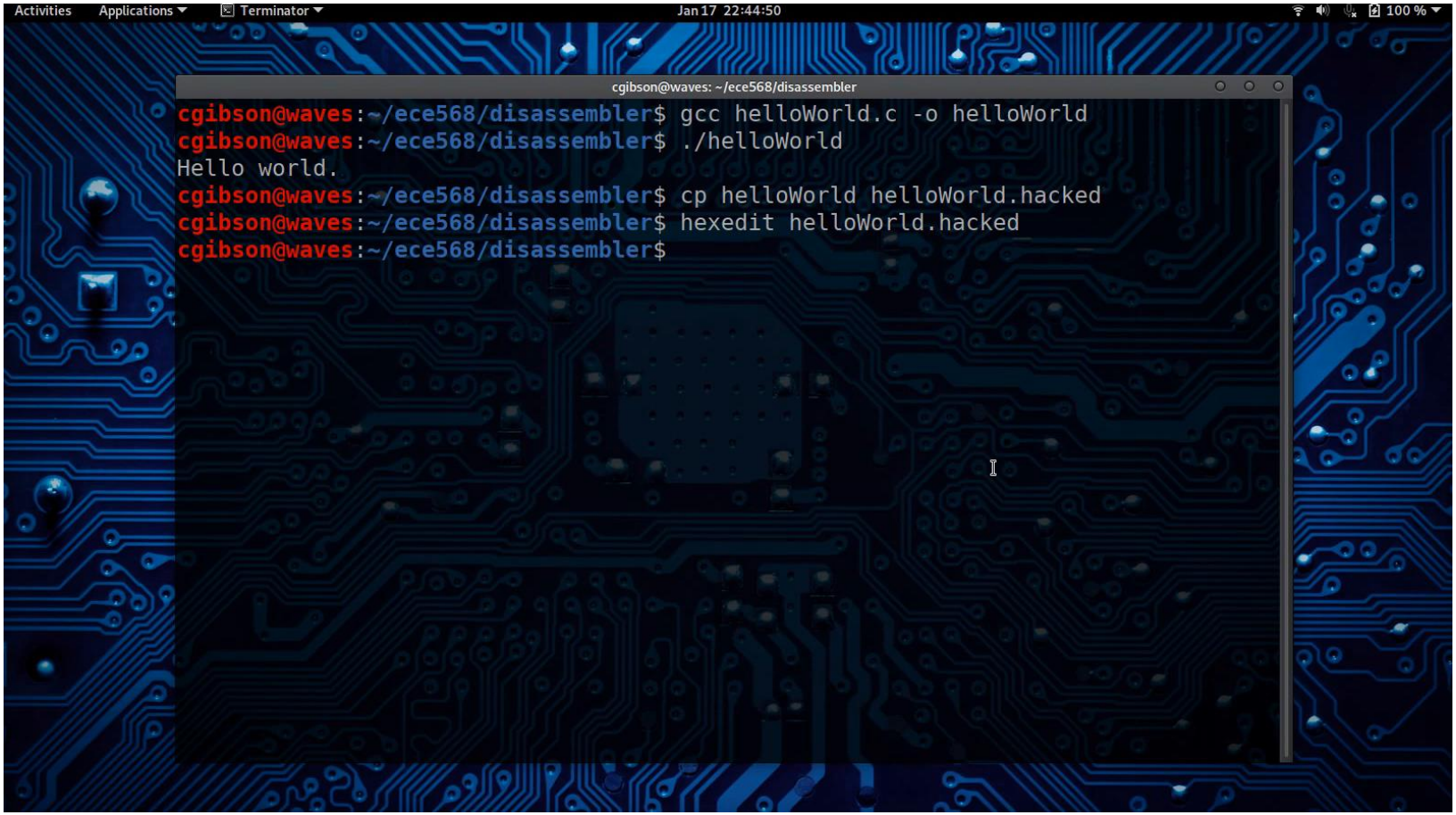

Activities
Applications
redasm
Jan 17 22:40:11
100 %
helloWorld — REDasm 2.0-20181231.7e961ae RC3
File
REDasm
?
Address
Symbol
00000000... _puts
00000000... _start
00000000... printHelloWorld
00000000... main
00000000... __libc_csu_init
00000000... __libc_csu_fini
00000000... _fini
Functions
Call Graph
Direction
Reference
--- jne loc_1181
Listing
Hex Dump
Segments
Imports
Exports
Strings
.text:000000000000107A hlt
.text:0000000000001080 deregister_tm_clones 4800002FA93D8D48
.text:00000000000010B0 register_tm_clones 4800002F793D8D48
.text:00000000000010F0 __do_global_dtors_aux 750000002F393D80
.text:0000000000001130 frame_dummy 894855FFFFFF7BE9
.text:0000000000001135 push rbp
.text:0000000000001136 mov rbp, rsp
.text:0000000000001139 lea rdi, str_2008 # STRING: "Hello world."
.text:0000000000001140 call _puts
.text:0000000000001145 mov eax, 5
.text:000000000000114A pop rbp
.text:000000000000114B ret
.text:000000000000114C push rbp
.text:000000000000114D mov rbp, rsp
.text:0000000000001150 sub rsp, 20
.text:0000000000001154 mov [rbp - 14 + local.2], edi
.text:0000000000001157 mov [rbp - 20 + local.4], rsi
.text:000000000000115B mov [rbp - 4 + local.0], 0
.text:0000000000001162 mov eax, 0
.text:0000000000001167 call printHelloWorld
.text:000000000000116C mov [rbp - 4 + local.0], eax
.text:000000000000116F cmp [rbp - 4 + local.0], 4
.text:0000000000001173 jne loc_1181
.text:0000000000001175 lea rdi, str_2018 # STRING: "Super-secret string... shhh..."
.text:000000000000117C call _puts
loc_1181:
.text:0000000000001181 mov eax, 0
.text:0000000000001186 leave
.text:0000000000001187 ret
.text:0000000000001190 push r15
.text:0000000000001192 lea r15, __frame_dummy_init_array_entry
.text:0000000000001199 push r14
Section".dynsym" contains a symbol table @ offset 0000000000000330
Section".symtab" contains a symbol table @ offset 0000000000003058
Found format 'ELF Format' with 'x86_64' instruction set
Function: main+27 Address: 0000000000001173 Offset: 0000000000001173 Segment: .text

```
                        ; ================== FUNCTION printHelloWorld ==================
.text:0000000000001135  push rbp
.text:0000000000001136  mov rbp, rsp
.text:0000000000001139  lea rdi, str_2008                   # STRING: "Hello world."
.text:0000000000001140  call _puts
.text:0000000000001145  mov eax, 5
.text:000000000000114A  pop rbp
.text:000000000000114B  ret
                        ; ================== FUNCTION main ==================
.text:000000000000114C  push rbp
.text:000000000000114D  mov rbp, rsp
.text:0000000000001150  sub rsp, 20
.text:0000000000001154  mov [rbp - 14 + local.2], edi
.text:0000000000001157  mov [rbp - 20 + local.4], rsi
.text:000000000000115B  mov [rbp - 4 + local.0], 0
.text:0000000000001162  mov eax, 0
.text:0000000000001167  call printHelloWorld
.text:000000000000116C  mov [rbp - 4 + local.0], eax
.text:000000000000116F  cmp [rbp - 4 + local.0], 4
.text:0000000000001173  jne loc_1181
.text:0000000000001175  lea rdi, str_2018                   # STRING: "Super-secret string... shhh..."
.text:000000000000117C  call _puts
                        loc_1181:
.text:0000000000001181  mov eax, 0
.text:0000000000001186  leave
.text:0000000000001187  ret
```


Activities
Applications
redasm
Jan 17 22:40:36
100 %
helloWorld — REDasm 2.0-20181231.7e961ae RC3
File REDasm ?
Address Symbol
00000000... _puts
00000000... _start
00000000... printHelloWorld
00000000... main
00000000... __libc_csu_init
00000000... __libc_csu_fini
00000000... _fini
Functions Call Graph
Direction Reference
--- jne loc_1181
Listing Hex Dump Segments Imports Exports Strings
00001060 8D 05 8A 01 00 00 48 8D 0D 23 01 00 00 48 8D 3D .....H..#...H.=
00001070 D8 00 00 00 FF 15 66 2F 00 00 F4 0F 1F 44 00 00 ......f/.....D..
00001080 48 8D 3D A9 2F 00 00 48 8D 05 A2 2F 00 00 48 39 H.=./..H.../..H9
00001090 F8 74 15 48 8B 05 3E 2F 00 00 48 85 C0 74 09 FF .t.H..>/..H..t..
000010A0 E0 0F 1F 80 00 00 00 00 C3 0F 1F 80 00 00 00 00 ....
000010B0 48 8D 3D 79 2F 00 00 48 8D 35 72 2F 00 00 48 29 H.=y/..H.5r/..H)
000010C0 FE 48 89 F0 48 C1 EE 3F 48 C1 F8 03 48 01 C6 48 .H..H..?H...H..H
000010D0 D1 FE 74 14 48 8B 05 15 2F 00 00 48 85 C0 74 08 ..t.H.../..H..t.
000010E0 FF E0 66 0F 1F 44 00 00 C3 0F 1F 80 00 00 00 00 ..f..D..........
000010F0 80 3D 39 2F 00 00 00 75 2F 55 48 83 3D F6 2E 00 .=9/...u/UH.=...
00001100 00 00 48 89 E5 74 0C 48 8B 3D 1A 2F 00 00 E8 2D ..H..t.H.=./...-
00001110 FF FF FF E8 68 FF FF FF C6 05 11 2F 00 00 01 5D ....h....../...]
00001120 C3 0F 1F 80 00 00 00 00 C3 0F 1F 80 00 00 00 00 ....
00001130 E9 7B FF FF FF 55 48 89 E5 48 8D 3D C8 0E 00 00 .{...UH..H.=....
00001140 E8 EB FE FF FF B8 05 00 00 00 5D C3 55 48 89 E5 ....].UH..
00001150 48 83 EC 20 89 7D EC 48 89 75 E0 C7 45 FC 00 00 H.. .}.H.u..E...
00001160 00 00 B8 00 00 00 00 E8 C9 FF FF FF 89 45 FC 83 ....E..
00001170 7D FC 04 75 0C 48 8D 3D 9C 0E 00 00 E8 AF FE FF }..u.H.=........
00001180 FF B8 00 00 00 00 C9 C3 0F 1F 84 00 00 00 00 00 ....
00001190 41 57 4C 8D 3D 4F 2C 00 00 41 56 49 89 D6 41 55 AWL.=O,..AVI..AU
000011A0 49 89 F5 41 54 41 89 FC 55 48 8D 2D 40 2C 00 00 I..ATA..UH.-@,..
000011B0 53 4C 29 FD 48 83 EC 08 E8 43 FE FF FF 48 C1 FD SL).H....C...H..
000011C0 03 74 1B 31 DB 0F 1F 00 4C 89 F2 4C 89 EE 44 89 .t.1....L..L..D.
000011D0 E7 41 FF 14 DF 48 83 C3 01 48 39 DD 75 EA 48 83 .A...H...H9.u.H.
000011E0 C4 08 5B 5D 41 5C 41 5D 41 5E 41 5F C3 0F 1F 00 ..[]A\A]A^A_....
000011F0 C3 00 00 00 48 83 EC 08 48 83 C4 08 C3 00 00 00 ....H...H.......
00001200 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001210 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001220 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001230 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001240 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001250 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001260 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001270 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001280 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
00001290 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ....
Section".dynsym" contains a symbol table @ offset 0000000000000330
Section".symtab" contains a symbol table @ offset 0000000000003058
Found format 'ELF Format' with 'x86_64' instruction set
Function: main+27 Address: 0000000000001173 Offset: 0000000000001173 Segment: .text


Activities
Applications
Terminator
Jan 17 22:42:36
100 %
cgibson@waves: ~/ece568/disassembler
cgibson@waves:~/ece568/disassembler$ gcc helloWorld.c -o helloWorld
cgibson@waves:~/ece568/disassembler$ ./helloWorld
Hello world.
cgibson@waves:~/ece568/disassembler$

```
                        ===================== FUNCTION printHelloWorld ====================
.text:0000000000001135  push rbp
.text:0000000000001136  mov rbp, rsp
.text:0000000000001139  lea rdi, str_2008                   # STRING: "Hello world."
.text:0000000000001140  call _puts
.text:0000000000001145  mov eax, 5
.text:000000000000114A  pop rbp
.text:000000000000114B  ret
                        ===================== FUNCTION main ===================
.text:000000000000114C  push rbp
.text:000000000000114D  mov rbp, rsp
.text:0000000000001150  sub rsp, 20
.text:0000000000001154  mov [rbp - 14 + local.2], edi
.text:0000000000001157  mov [rbp - 20 + local.4], rsi
.text:000000000000115B  mov [rbp - 4 + local.0], 0
.text:0000000000001162  mov eax, 0
.text:0000000000001167  call printHelloWorld
.text:000000000000116C  mov [rbp - 4 + local.0], eax
.text:000000000000116F  cmp [rbp - 4 + local.0], 4
.text:0000000000001173  nop
.text:0000000000001174  nop
.text:0000000000001175  lea rdi, str_2018                   # STRING: "Super-secret string... shhh..."
.text:000000000000117C  call _puts
.text:0000000000001181  mov eax, 0
.text:0000000000001186  leave
.text:0000000000001187  ret
```


Activities
Applications
Terminator
Jan 17 22:44:50
100 %
cgibson@waves: ~/ece568/disassembler
cgibson@waves:~/ece568/disassembler$ gcc helloWorld.c -o helloWorld
cgibson@waves:~/ece568/disassembler$ ./helloWorld
Hello world.
cgibson@waves:~/ece568/disassembler$ cp helloWorld helloWorld.hacked
cgibson@waves:~/ece568/disassembler$ hexedit helloWorld.hacked
cgibson@waves:~/ece568/disassembler$

```
#include <stdio.h>

unsigned int printHelloWorld()
{
        printf("Hello world.\n");
        return(5);
}

int main(int argc, char * argv[])
{
        unsigned int result = 0;
        result = printHelloWorld();
        if(result == 4) {
                printf("Super-secret string... shhh...\n");
        }
        return(0);
}
```

Reverse Engineering Tools

Disassembler / Runtime analysis:

- **IDA Pro:** Commercial
- **xdbg64:** Windows

Decompilers:

- **Snowman:** Compiled code to C++

Defenses

Buffer Overflow Defenses

Many of the attacks discussed have depended on overflowing buffers. The most obvious way to defend against buffer overflow vulnerabilities is not to make them:

- Audit code rigourously
- Use a type-safe language with bounds checking
  - *e.g.*, Java, C#
  - Code will be **memory safe**: compiler will enforce the memory access rules of the language

However, this is not always possible:

- Too much legacy code
- Source code is not available
- Performance may be a concern
- Easy to write C code without correct checks

Other Options for Defense?

Buffer overflow attack requires an input string to be copied into a buffer without bounds checking

- Typical attack requires three steps
  - Control over a location such as return address
  - Overwrite location with guessed address
  - Inject and execute shell code

What is needed for these steps to succeed?

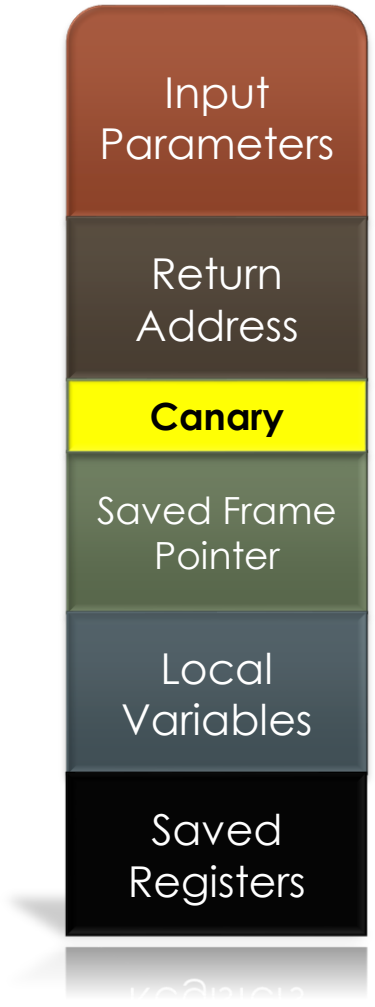
- Return address overwrite
- Target address has to be guessed
- Injected code has to be executable

Let's look at how to detect or prevent each of these steps…

Defending Against Stack Smashing

Recent protection techniques will prevent the return address from being overwritten

- **Stackshield**
  - Put return addresses on a separate stack with no other data buffers there
- **Stackguard**
  - On a function call, a random **canary** value is placed just before the return address
  - Just before the function returns, the code checks the canary value and, if the value has changed, the program is halted
  - MS VC++ compiler supports it with the *GS flag*
  - Recent GCC compilers support it
  - Does the canary stop format string attacks?



Defending Against Stack Smashing

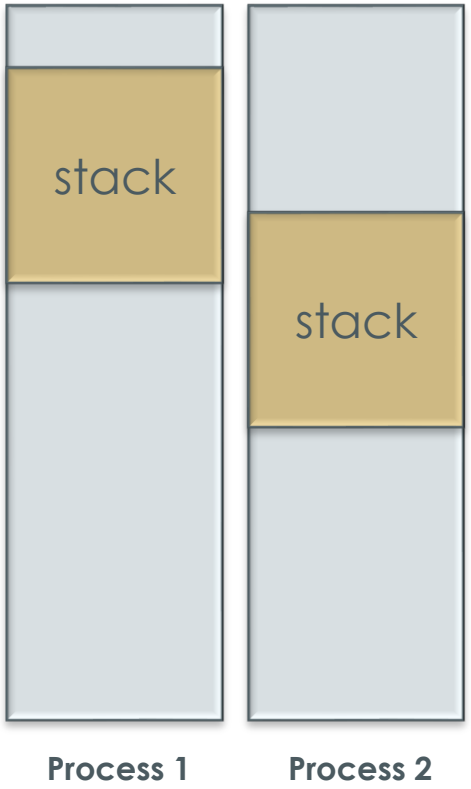
Run Time Checking: Libsafe (Avaya Labs)

- Dynamically loaded library
  - Overrids **libc.so** (usually via /etc/ld.so.preload)
  - Done at runtime, so doesn't need program recompilation or source-code changes
- Intercepts calls to dangerous functions such as **`strcpy`**
  - Validates sufficient space in current stack frame

Address-Space Layout Randomization (ASLR)

Recall that the target address (*e.g.*, the buffer's location on the stack) has to be guessed:

- With ASLR, the OS maps the stack of each process at a **randomly** selected location with each invocation
  - An attacker will not be able to easily guess the target address
  - Application will crash rather than executing the attacker's code
  - ASLR also randomizes location of dynamically loaded libraries, making it harder to perform return-into-libc attacks or GOT overwrites
- Linux 2.6 and Windows Vista use ASLR



Non-Executable Pages (NX)

- If stack is made non-executable, then shellcode on the stack will not execute
  - Recent Intel, AMD processors allow non-executable pages
    - Page tables have NX protection bit
  - Requires support from OS
    - NX implemented in Windows XP SP2 patch
- However, non-injection attacks are still possible
  - E.g., return-into-libc attacks, argument overwrite attacks

Static Analysis

Analyzes code for common errors or issues that could lead to vulnerabilities.

- **LGTM:** Automated scans of GitHub
- **Flawfinder:** C/C++ source code analysis

https://lgtm.com
https://dwheeler.com/flawfinder

Vulnerability Databases

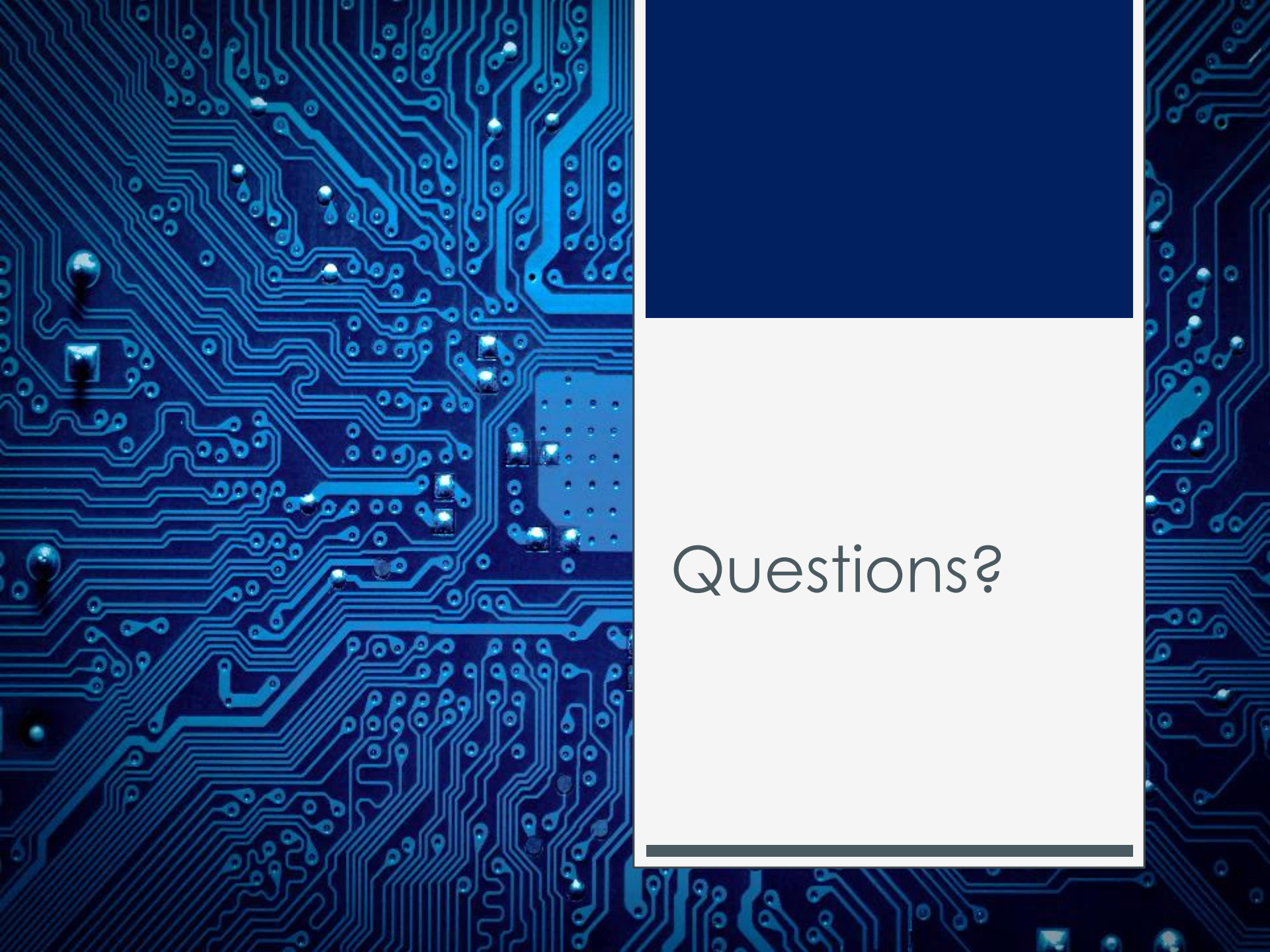
To aid computer administrators, there are several large databases of vulnerabilities on the Internet:

- **National Vulnerability Database:** http://nvd.nist.gov
- **CERT:** http://www.cert.org
- **SecurityFocus:** http://www.securityfocus.com/vulnerabilities
- **Bugtraq:** http://www.securityfocus.com/archive/
- **OSVDB:** http://www.osvdb.org

For any program and version, one can query these databases and get a description of the vulnerability

Conclusion

- Easy to make a mistake, end up with a vulnerability
  - Exploiting them takes a bit of work, but is not beyond someone who knows what they are doing
- Certain vulnerabilities can be removed by moving to safer languages
  - A lot of vulnerabilities result from uses of pointers and running off the end of arrays
- However, the only real defense is to be aware of what vulnerabilities exist, to be extra careful when creating code and let others audit your code


Questions?