332:424 – Introduction to Information and Network Security

Project – MIP-BPI: Malicious IDAPro Byte-Patching Interface

Analysis of IDAPro & IDAPython to Manipulate Executables

Groupname – My Little Packet

By David Lambropoulos, Harsh Shah, Demetrios Lambropoulos, Lola Castellanos

Professor Saman Zonouz
December 4, 2015

Rutgers University
School of Engineering

Table of Contents

Abstract

As the growth in size and complexity of today's programs, analysis and manipulations of these programs have become increasingly difficult. IDA Pro offers useful resources for executable file disassembly, as opposed to competitors and predecessors, due to its robust interface and its ability to provide in-depth analysis tools. There are new techniques that effectively reduce effort on the user and expedite analysis; namely, the development of a plug-in that will change the way the user approaches the problem. MIP-BPI will offer the user simplicity and ease of use while still providing powerful functions to the user. Through the investigation of malicious byte-patching and methods of obtaining useful information, we have developed MIP-BPI to offer the user information on the executable file and means to edit it. MIP-BPI can be efficient at extracting information from the disassembled executables.
Keywords: byte-patching, executable, disassemble, IDAPro, program

1 Introduction

In order to introduce MIP-BPI we refer to the following scenarios. If an analyst was given a malicious executable, how can he test all possible results the malicious program can have when executed? At execution time, malware will adapt to its host and it will display the functionality based on its environment. For the analyst this can be cumbersome as what malware does through one path of execution may not be its full malicious capability. To remediate this issue, byte patching can be an answer. If the analyst can identify the location where malware chooses to behave in a certain way, the analyst can byte patch this decision, and for the malware to behave a different way. Ideally, the analyst would repeat this process to understand the full functional capabilities of the malware. Futhermore, consider an analyst who harbors malicious intent. He may want to bypass or alter a legitimate program and alter it for malicious purposes. Consider the case of installing a program which requires a paid license for use. The malicious analyst may want to over-write or skip installation checks and force the program to be installed through unauthorized methods. Even more primitive than these examples, a programmer may want to identify and alter targeted bytes within a file. We present our plug-in MPI-BPI as a solution to the problems listed above.

MIP-BPI is a plug-in for IDAPro that was programmed in Python and uses simple IDC commands to analyze disassembled code. This plug-in is 'Malicious IDAPro Byte-Patching Interface' or dubbed MIP-BPI for short. MIP-BPI is a simple GUI that allows the user to make simple calls and learn about their disassembled code without making them go through the hassle of physically looking through it themselves and above all, simply byte-patch or inject code in the program.

Groups in the past were able to successfully byte-patch executables but the process was very tedious and not very simple to do for the inexperienced. However, our solution, as proposed, will have the ability to offer the user many dynamic ways of maliciously or non-maliciously manipulating an executable to fit their liking. Additionally, with our program, the user will be able to perform multiple tasks with one single plug-in which has not been previously created nor recorded in any documentation found. This GUI will give the user the ability to learn all facets of the executable at hand and then take what they learned and apply it in such a way that they might be able to accomplish the tasks they set out to accomplish.

2 Background

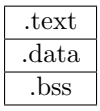
2.1 Executables: Programs & Processes

Traditionally, an executable or program is a set of instructions that are designed to perform a designated task. An executable in C is generated by the following process:

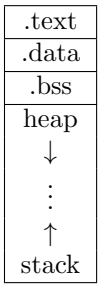
$$\underbrace{.c}_{\text{Source Code}} \rightarrow gcc \rightarrow \underbrace{.s}_{\text{Assembly Code}} \rightarrow assembler \rightarrow$$

$$\underbrace{.o}_{\text{Object Code}} \rightarrow linker \rightarrow executable$$

A program is composed of three main parts: 1) .text, 2) .data, 3) .bss. Where the text section contains the executable code, the data section contains static data such as variables that are initialized at compile time, and finally the block started by symbol or bss contains static variables uninitialized at compile time. This can be visualized as the following structure,



An executable that is executing is known as a process which is an operation that takes the given instructions and perform the manipulations as per the code. A process is similar to an executable in that it has .text, .data, and .bss contained within it but also a dynamically allocated heap and stack that grows/shrinks as needed. This can be visualized as the following,



Using this information, one can understand how an executable is created and how the sections are organized within. This is helpful when looking into an executable and changing code for malicious intent.

2.2 IDAPro Overview

IDAPro is a popular disassembler provided by Hex-Rays. It is a static analysis tool that essentially receives an input executable file and outputs a mapping of the assembly instructions within the binary file. It is a very useful tool that offers a lot of features for tracing instructions and data across a binary file. When one opens the application in IDAPro it disassembles the binary file. One is then presented with many views and options right away. Moreover, one will have a multiplicity of choices on how to view the instructions and access to the default views included in IDA View or Hex-View. The IDA-View is a listed code segment of multiple sections of the binary (i.e., data, idata, text, etc). The Hex-View, as its name suggests, is a listed hexadecimal view of the binary file. Other default views on IDA are the imports and exports of the binary file. The imports tab presents the runtime libraries and the functions the binary calls in order to operate. This is very important for the static analysis of malware as it may hold evidence of malicious behavior (i.e., calls to functions that do not have anything to do with the intent of the program). The exports tab present the functions the binary is exporting, this is not a typical behavior of non-malicious executables. Other functional behaviors of IDA pro include tagging either variables known to exist throughout the instructions or tracing a variable throughout the execution of the instructions. For example, Data XREF, is used to track where items in the data section are accessed in the instructions. Similarly, in IDA Pro, functions can be cross-referenced to see where and when they are called in the binary. If one were to disassemble a hello world program, they might expect to see something like that of figure-1.

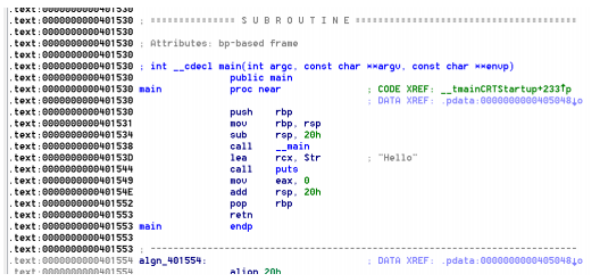


Figure-1: Hello World disassembled in IDAPro

2.2.1 IDAPro as a Dis-assembler

IDA Pro has the ability to explore binary programs, for which source code is not available and creates maps of the execution paths. Since assembly language is not always a user friendly language to read.

2.2.2 IDAPro as a Debugger

IDAPro allows an analyst to single-step through the code that they are investigating. Also, it helps obtain data that a more powerful static disassembler will be able to process in depth. Finally, it adds a layer of security because with a remote debugger one is able to safely dissect potentially harmful programs.

2.2.3 IDAPro as a Programming Environment

IDAPro contains a complete development environment that consists of a very powerful macro-like language. Using languages like C or Python a developer can develop plug-ins to enhance IDAPro's functionality.

2.2.4 Hostile Code Analysis

IDAPro is a standard in the field of hostile code analysis and for a notable example was used in the 2001 CODE RED incident when the worm eEYEs payload targeted the white house. The IDAPro software was deployed to analyze and understand the worm.

2.2.5 Vulnerability Research

IDAPro can be used to help detect vulnerabilities which can be xed before exploiters notice them and take advantage of them.

2.3 IDAPython Overview

2.3.1 Description

IDAPython is the fusion of the power and ease of use of Python with C-like scripting language that is IDC (IDAPro's scripting language that aids in analysis automation).

2.3.2 Examples: Plug-ins and More

If one wanted to display all of the code segments within a program, one could write a Python script such as,

```
import idautils

for seg in idautils.Segments():
    print idc.SegName(seg), idc.segStart(seg)
    , idc.segEnd(seg)
```

Code-1: Print Segments IDAPython Script

Upon running this script on the disassembled code of hello.exe, one would obtain the list of the following segments as shown in Figure-2.

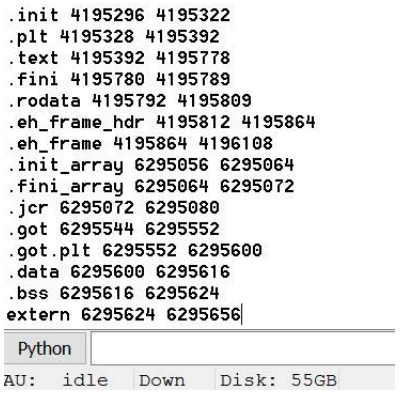


Figure-2: Segments of HelloWorld.c

If one wanted to display all of the functions that occur within a program, one could write a Python script such as,

```
import idautils

for func in idautils.Functions():
    print hex(func), idc.GetFunctionName(func
    )
```

Code-2: Print Functions IDAPython Script

2.4 Overview of Byte-Patching

To patch something essentially means just to modify it. Byte-patching is the art of individually overwriting a byte in a binary executable with one that is different than that of the original. This does not have to be used for malicious intent although it can be used to do so as well. A form of malicious byte-patching can come in the form of an email containing an update for your OS or a software on your computer and the attached software update seems to be legitimate but has been patched to actually perform actions that cause catastrophic effects to occur to your machine and to the machines of others.

2.5 Overview of Code Caves

A code cave is an unused block of memory that can be used to inject instructions into. These are usually formed when more space is allocated than necessary. Example of this is that if 5 bytes were allocated and 3 bytes were used there are then 2 bytes remaining that could be used by a malicious process to inject bytes. In IDA Pro these are usually represented by "db 00". These are often exploited by hackers to execute their arbitrary code in the compiled program. This is easy and fast for coder to modify a process and is can be very efficient if used correctly. However, if used incorrectly, one can easily break a program if the program had already been using that code cave and you overwrite the existing script and break the control flow of the program and the lack of space in these caves in which to inject code into.

2.6 Overview of Code Injection

Code injection in essentially the art of injected jump statements into the code and then continuing to those jumped locations, usually code caves, and performing the statement wanted by the injector and usually jumping back to the original control flow.

3 Overview of MIP-BPI

3.1 Description

MIP-BPI is a IDA Pro plug-in, which will grant the user a simplified interface to retrieve basic information from an executable file and also edit targeted hex code segments with new instructions. The user will be presented with a graphical user interface, which will contain buttons that designate the possible uses of the plug-in. The user will choose a button and will fill in the appropriate fields that can be populated and then execute the task. The current possible uses of MIP-BPI are to retrieve information about the executable file, byte patch over a particular address space, and inject bytes into particular address space. We explore these further in the next section.

3.2 Lo-Fi Prototyping

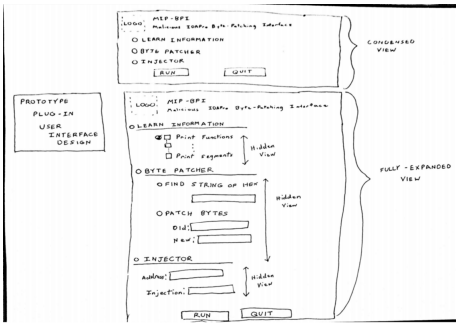


Figure-3: Prototype Plug-in User Interface Design

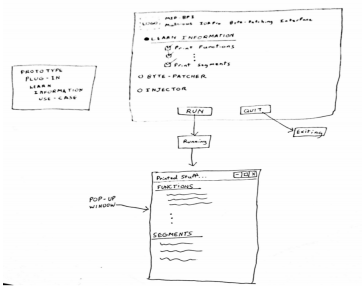


Figure-4: Prototype Plug-in Learn Information

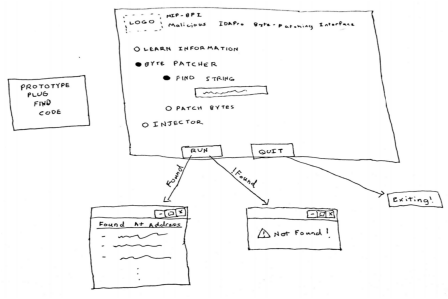


Figure-5: Prototype Plug-in, Find Code

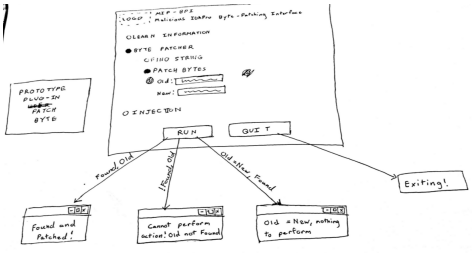


Figure-6: Prototype Plug-in, Patch Bytes

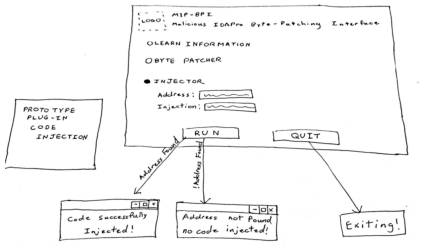


Figure-7: Prototype Plug-in, Code Injection

3.3 Uses

Looking at Figure-3 we see a prototype of the view the user is presented with on start of the plug-in. Starting with a condensed view, the user is presented with three choices as stated earlier. When the user either picks Learn Information, Byte Patcher, or Injector, the condensed view is expanded and the user is prompted to enter fields that correspond to each option.

To explore these options further, lets start by taking a closer look at Learn Information in Figure-4. When the user chooses Learn Information, he will be able to see information including but not limited to the functions called in the executable and the address ranges of the different segments in the executable file. When the user has finished selecting which information he would like to observe, the user will click run and will be presented with a pop-up containing the information he has chosen to view. The user can use the information presented here to engage in the next two use cases.

Figure-5 depicts the case where the user chooses to byte patch an executable file. On screen, the user will be prompted with two options: one to find the addresses occupied by a target string and two a short form to replace old bytes with new bytes on a certain range of addresses. When the user enters target string, a window will pop up either detailing all locations the target string is found or will notify the user the target string does not exist in any segment of the executable. If the string is found, the user will be armed with the information to byte patch the string.

Figure-6 illustrates how the user will byte patch using MIP-BPI. To byte patch a string, the user will enter the bytes in the given address range of the old string in the text field marked old followed by entering a the new replacement group of bytes in the new text field. Once the user has entered the pertinent information and has clicked run one of the following notifications will be presented to the user. The bytes will be patched and the user will be notified of success, the old bytes will not be found and the user will notified the patch failed, and last, if the user did not change the bytes in the new field he will be notified that no change was placed on the executable.

Figure-7 shows the use of Injector in MIP-BPI. The injector will receive a series of bytes to inject at a particular address and place them starting at the specified address. If the injection is successful the user will be notified of the success or if the injection fails due to the address not being found, the user will be notified of the failure. With this high level description satisfied we now move to a detailed description of the implementation.

4 Analysis of MIP-BPI

4.1 Algorithms

4.1.1 Print Code Caves

```
#Print Code Caves
def printCaves():
    # Get the starting address of the program
     and the endding address
    beginning = idc.MinEA()
    end = idc.MaxEA()

    # Start the current address at the
    beginning of the program
    current = beginning

    i = 0

    # Iterate through the program finding
    code caves
    while current <= end:
        # Check if the disassembly at the
    line is empty
        if (idc.GetDisasm(current) == "db
    0"):
            # Record the start of the cave
            startAddr = current
            # Variable to record the number
    of lines available in code cave
            size = 0
            # Iterate to the end of the code
    cave while checking that the edge
            # of the program has not been
    reached
            while (idc.GetDisasm(current) ==
    "db    0"):
                # Increment the size of the
    cave by 1
                size = size + 1
                # Increment to the next
    address
                current = current + 1
            # Mark the end of the cave as the
     address at the end of the loop
            endAddr = current
            # Append the cave information to
    the list of caves
            print "Cave " + str(i) + ": "
            print "--------------------"
            print "Start: " + str(startAddr)
            print "End: " + str(endAddr)
            print "size: " + str(size)
            print "\n"
            i = i + 1
        # Go to the next address in the
    binary
        current = current + 1
```

Code-3: Print Code Caves IDAPython Script

4.1.2 Create HEX String

```
# Hexitize me captain
# (Not it use)
def hexitize(inputStr):

    # Convert string into its hex equivalent
    newStr = inputStr.encode("hex")

    return ' '.join(newStr[i:i+2] for i in
    xrange(0, len(newStr), 2))
```

Code-4: Hex String IDAPython Script

4.1.3 Byte-Patching

```
# The patch-inator
def patchy_watchies(original, changed):
    hexStr = hexitize(changed)
    print "Hex: " + hexStr + "\n"
    print "Original: " + original + "\n"
    # Iterate through the segments of the binary
    for seg in idautils.Segments():
        # Segment name of the current segment
        segmentName = idc.SegName(seg)
        # Start address of the segment
        startAddr = idc.SegStart(seg)
        # Current address of the segment
        cur_addr = idc.SegStart(seg)
        # End address of the segment
        endAddr = idc.SegEnd(seg)
        # Check if the current segment is the
    segment for raw data
        if segmentName == ".rdata":
            # Iterate through the segment
            while cur_addr <= endAddr:
                # Check to see if the current
    address contains a String
                if type(idc.GetString(cur_addr))
    == type("String"):
                    # Check if the String contains
     the user request
                    if idc.GetString(cur_addr).
    find(usrin) != -1:
                        for i, c in enumerate(
    usrchange):
                            print "patching byte"
                            idc.PatchByte(cur_addr
    +i, int(c.encode("hex"), 16))
                        break#for i, c in
    enumerate(str):
                        #   idc.PatchByte(cur_addr
    +i, Byte(int(c, 16)))
                    # idc.PatchByte(cur_addr,
    ord(usrchange.encode("hex"), 16))
                # Move cursor to next head
                cur_addr = idc.NextHead(cur_addr,
    endAddr)
```

Code-5: Byte-Patching IDAPython Script

4.2 Outlook

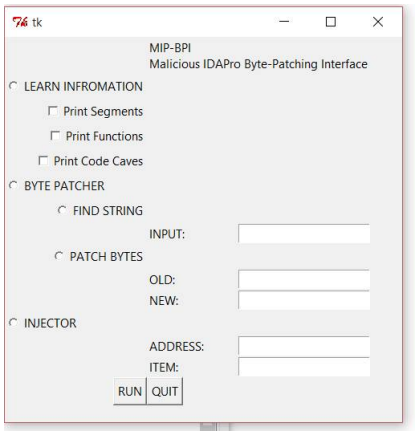


Figure-8: MIP-BPI Plug-in

4.3 Code Explanations

4.3.1 Printing Code Caves

It begins by first getting the starting address of the program. It the starts by setting the current address to the beginning of the program by using what it previously just obtained. It then begins to iterate through the program finding the code caves by checking if the disassembly at each line is empty or equal to "db 0". If it is it records that address as the start of the cave and begins to iterate to the end of the code cave while checking that the edge of the program has not been reached. It proceed to loop until it reaches another "db 0" increasing the size and the current position by 1 each time. Once done with that loop it marks the end of the cave as the address at the end of the loop and prints the information of starting address, ending address, and size of cave to the user. It then begins to proceeding its walk through the program and continuing the same process.

4.3.2 Byte-patching

It begins by taking in its arguments as the original code and the code in which the original is to be changed to. It then begin to iterate through the segments of the binary for every segment in the programs segments. It then gets the name of the segment and the start address and current address (also equal to start) and the end address of the segment. It then begins to check if the current segment is the segment for raw data and if it is it will continue to iterate through the segment and check to see if the current address contains a string and if that is true it will check if the string contains the users request and that is true it will loop through and patch all the individual bytes and will then exit.

4.4 Problems Encountered

We observed that there exists some issues when using Tkinter to construct the GUI in Python and when running that constructed GUI in IDAPython you need to put the thread to sleep for 4 seconds before allowing it to build or else it will leave your program in a frozen state that can only be remedied by actually restarting IDAPro. No further information has been gathered on this issue however we believe that this is a rendering issue.

If a user enters an array of bytes longer than that of the original array of bytes this will result in a buffer overflow and start overwriting bytes outside of the scope in which the user initially intended.

5 Experimental Setup

5.1 Environment

To build the MIP-BPI plug-in we used Python 2.7 on a Windows 10 machine. In the Python code we utilized the Tkinter library, which is a GUI library in python, to build the interface as well as the IDC libraries provided by IDA to build the functionality of the plug-in. To test MIP-BPI, we generated sample "Hello World" program executable files on Windows 10, Linux Mint 17.2 Rafael, and Ubuntu 14.04 Trusty Tahr. For all testing of MIP-BPI we byte patched our sample "hello world" executables in IDA Pro version 6.6 (64-bit).

6 Related Work

IDA pro is meant to provide the user with the detailed behavior of a binary by offering dis-assembly tools. It offers byte patching, but it is not designed to make it easy for the user to modify the binaries. In order to perform binary patching users need to figure out what each part of the binary does, also where and how it performs the actions that are to be modified. In order to perform byte patching with IDApro the user needs to first edit a configuration file, then edit the IDA database that contains the real binary, which is not the original binary. The next steps include generating the DIFF file through procedure, which will enumerate what and where changes happened, and lastly download and compile the IDA patcher.c that will perform the byte patching.

OllyDbg is also a debugger useful for tracing registers, recognizing procedures, tables, constants, strings. It also offers third party plugins, object file scanning, and it loads Dlls, among other functions. We recognize that OllyDbg may also be used in a similar capacity to IDA Pro to disassemble and patch binary files as it includes features such as adding assembly code to a direct address or over-writing user selected bytes. To patch instructions in OllyDbg, one needs to find the target instructions by stepping through the executable file. Once the instruction is found, by double clicking on it, the user will be presented with a dialog box that asks the user if he wished to assemble a new instruction at the address. The user can choose to manually enter a new instruction at this time. If the user wants see the result of the modification, he simply clicks play and the modified file will be run. We also recognize that although OllyDbg offers a slightly faster mechanism for manual instruction patching as opposed to IDA Pro, the uses of these programs and robustness vary greatly. For a user to byte patch manually through OllyDbg would be inefficient and slow. Through the use of IDA-Pro's robust interface and availability of information the user, the user will have an easier time determining the segments to patch through IDA Pro. Then through using a plugin for patching will be able to quickly exercise the patches.

7 Conclusion

It has become increasingly important to find easier and faster ways to analyze and to modify code. IDA pro was created for the user to comprehend the code, but not to work directly with it. MIP-BPI has been created to contribute to the easiness of understanding and modifying code when using IDA pro. It offers the user an easy way to analyze and manipulate a binary without having the user navigate through multiple software. MIP-BPI offers the user a method to quickly leverage addresses of target Hex code segments to be byte patched. As it stands, MIP-BPI can be used to quickly display the address of a target string in an executable file and re-write the bytes of a targeted address within the executable by the user.

8 Future Work

In future implementations of MIP-BPI we want to be able to leverage code caves to alter the functionality of a program. A code cave is the redirection of a program to execution of a remote set of instructions and back to the set of instructions from which the program originally left. Much like a function call, a code cave is a set of instructions that will be executed given certain parameters, however, the difference is while functions are written in the source code of a program, code caves are written in after the program has been created or when the source code is not available. Code caves can be exploited through either a linked library or injection of assembly directly into the executable.If the users chosen methodology to leverage a code cave is through a dll, then the user must explicitly link the dll to the same process as the program. If the user chooses to leverage a code cave through assembly injection, the user must find suitable instructions to overwrite. Common practice has been to find segments in the executable where no work is being done by the program; for example, through segments where all the instructions are DB 00, or no operation. Since

the purpose of MIP-BPI is to work directly with and alter the users executable file, we wish to use the latter method. We will extend the functionality of MIP-BPI to find code caves in the executable and return the address of these segments to the user. The user can leverage these segments to inject code into the executable without harming the regular control flow of the program. Based on the user's desired code injection, we will extend the functionality of MIP-BPI to determine if an injection can fit within the executable file.

9 References

[1] Applied Cracking and Byte Patching with IDA Pro. *http://resources.infosecinstitute.com/applied-cracking-byte-patching-ida-pro/*

[2] Eagle, Chris. IDA PRO Book, The Unofficial Guide to the Worlds's most popular Dissasembler.

[3] How to Patch Binary with IDA pro. *http://marcoramilli.blogspot.com/2011/01/how-to-patch-binary-with-ida-pro.html*

[4] Injeting a DLL prior to process execution. *http://newgre.net/node/4*

[5] OllyDbg. *http://www.ollydbg.de*

[6] The begineers Guide to Codecaves. *http://www.codeproject.com/Articles/20240/The-Beginners-Guide-to-Codecaves*