# Automatic Binary Patch Transplant - Final Report

Omkar Bhat (odb6pz), Jack Verrier (gjv7qw), Zoya Yeprem (zy8ez)

## Introduction

Vulnerabilities in software threaten the safety of computer programs and therefore many software security researchers use patches to overcome this problem. Patches are usually generated with human intervention, which is very time-consuming and often causes numerous problems. Therefore there is a need to find ways to automatically transplant patches. Currently, there is no algorithmic way to automatically transplant patches across different binaries, and this active research area motivates us to investigate further as a course project.

We made some assumptions to narrow down the scope of the original problem to a reasonable size for the purpose of this course project. These assumptions include:

- The selected infrastructure is Linux.
- Binaries used are unstripped. this gives us access to important information like function names which are useful when finding a location for the patch to be transplanted.
- No optimization; If optimization is turned on the compiler, it might show us optimized binaries which have a few modifications for faster runtime which might be treated as false positives for our analysis.
- The patch is assumed to be simple because the introduction of a single condition like “if” adds a few lines of code in the binary. Add a few lines of code means that we need to handle multiple things like a basic increase in the size of the binary. An increase in the size of the patched function results in changes in the relative address as well as the function locations and data locations. We also need to modify the ELF Headers accordingly and make sure that new lines of code do not overwrite any part of the code.

## Contributions

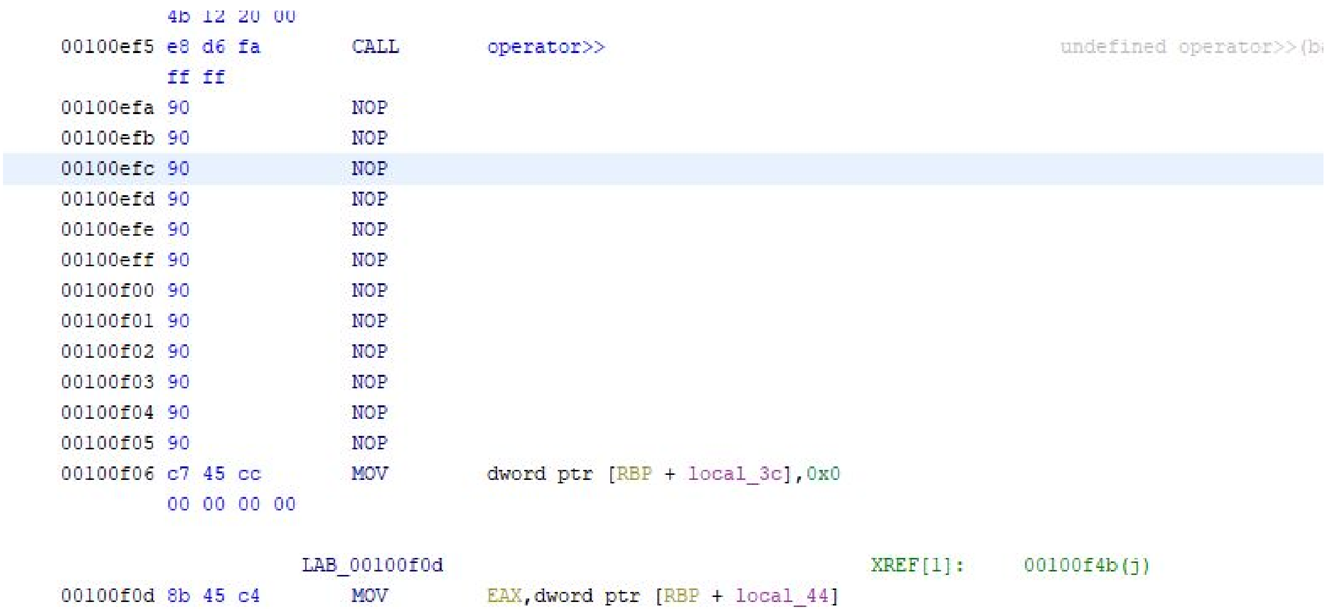
We approached this problem by first looking into the manual patch transplant process to try to learn what changes in the binary during a patch transplant. To do so we created two versions of a toy program - “Merge Sort” - one with security vulnerability and one without. Then tried to transplant the security patch from one to another. To do so we manually looked into two binaries and found the differences, then copied the differences to the vulnerable binary. What we learned from this process was that adding a minor “if” statement changes many things in the binary like change in data memory locations, indirect address, the function itself, address and the size of the binary, change in Headers, alignment and introduction of branches. We used this knowledge to apply the same patch to a different binary file: QuickSort. What we have learned from this is that the changes in bytes are correlated, this means the changes in the binary still depends on characteristics of the patch like the size, its functionality, etc. Making the relevant changes to the other locations in the binary file is not an active problem as it can be done with a good degree of accuracy.

Consider a patch with the introduction of a new “if” statement. To accomplish compilation after inserting this code, we either need to change the ELF metadata and code references or pad the source with enough NOPs such that we have sufficient space in the binary to add our code without needing to change values elsewhere in the binaries. Padding the source code in such a way that we can replace the padding with the patch means that we need the source code to make sure that this works perfectly. But in a real-life situation, we won't be having a source code for the binary that we are supposed to patch.

Also, there is a way that can be used when applying a patch to make patching a little easier. We need to make the patch as small as possible without losing the intended effect. This means we can actually effectively convert the patch into a compressed version which is then easier to transplant to the new binary that needs to be patched. To achieve this we need to reduce the number of dependencies the patch has on the previous code as this will mean that we changing something doesn’t tamper with the actions in the previous code.
As you can see from the following images adding padding of NOPs helps us as we can just replace those bytes with the bytes of the patch. This will help us patch easily.

**<u>Toy Binary:</u>**

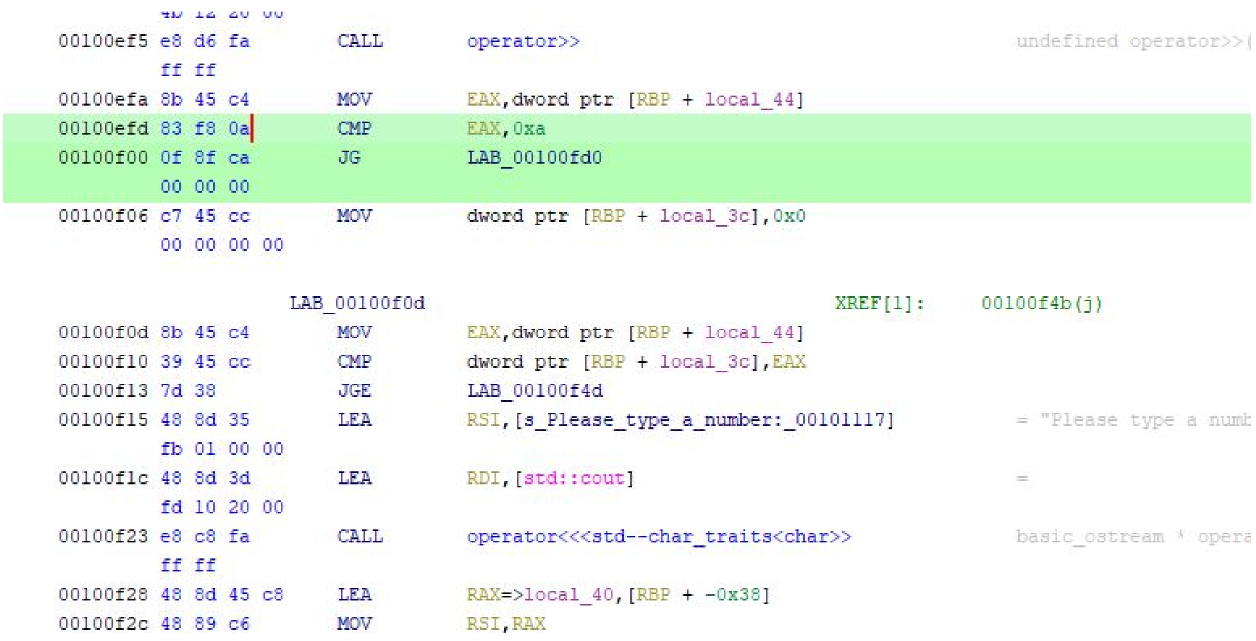
Useless Code (Adding NOPs):



```
          4b 12 20 00
00100ef5 e8 d6 fa       CALL      operator>>                                undefined operator>>(b
         ff ff
00100efa 90             NOP
00100efb 90             NOP
00100efc 90             NOP
00100efd 90             NOP
00100efe 90             NOP
00100eff 90             NOP
00100f00 90             NOP
00100f01 90             NOP
00100f02 90             NOP
00100f03 90             NOP
00100f04 90             NOP
00100f05 90             NOP
00100f06 c7 45 cc       MOV       dword ptr [RBP + local_3c],0x0
         00 00 00 00

                     LAB_00100f0d                          XREF[1]:     00100f4b(j)
00100f0d 8b 45 c4       MOV       EAX,dword ptr [RBP + local_44]
```

Patch:



```
         4b 12 20 00
00100ef5 e8 d6 fa       CALL      operator>>                                undefined operator>>(
         ff ff
00100efa 8b 45 c4       MOV       EAX,dword ptr [RBP + local_44]
00100efd 83 f8 0a       CMP       EAX,0xa
00100f00 0f 8f ca       JG        LAB_00100fd0
         00 00 00
00100f06 c7 45 cc       MOV       dword ptr [RBP + local_3c],0x0
         00 00 00 00

                     LAB_00100f0d                          XREF[1]:     00100f4b(j)
00100f0d 8b 45 c4       MOV       EAX,dword ptr [RBP + local_44]
00100f10 39 45 cc       CMP       dword ptr [RBP + local_3c],EAX
00100f13 7d 38          JGE       LAB_00100f4d
00100f15 48 8d 35       LEA       RSI,[s_Please_type_a_number:_00101117]    = "Please type a numb
         fb 01 00 00
00100f1c 48 8d 3d       LEA       RDI,[std::cout]                           =
         fd 10 20 00
00100f23 e8 c8 fa       CALL      operator<<<std--char_traits<char>>        basic_ostream * opera
         ff ff
00100f28 48 8d 45 c8    LEA       RAX=>local_40,[RBP + -0x38]
00100f2c 48 89 c6       MOV       RSI,RAX
```

Next, we took what we learned from this simple patch transplant and we tried applying it to a real-world problem - transplanting patches across different binaries for different ls versions.

**Real-world Binary:**

Now considering a binary file B (with vulnerability), we will add several redundant code lines such that its presence doesn't affect the normal processing of the code. This code will be replaced by the patch, this allows us to mainly focus on applying the patch rather than worrying about ELF header changes, the effect on other functions and the data, and indirect address changes.

**Challenges**

- Updating indirect address, jump, function address, size, and alignment.
- Changing things in the Header section. Things like version, entry point address flags, size of section headers, size of program headers, the start of the program headers.
- Reducing the size of the patch.
- Add padding to the area that needs to be patched.
- Active Research Area

**Things we tried**

1. Adding a single statement as a patch.
   To start, we added one single statement to the binary. We kept the initial patch simple enough to allow us to monitor the changes while making sure the program will continue to work as intended.
2. Adding an if-else statement as a patch.
   Next, we added two patches: one for an “if” statement, whereas the other is for the “else” statement. This complicates the patch transplant process since we need to apply a series of patches while making sure that the program still works as intended.
3. Adding multiple lines as a patch.
   Adding bigger patches means we need to shift everything that is relevant by the size of the patch.This may lead to discrepancies and other things like address are not found or data that is found is not correctly aligned.
4. Replacing a set of useless code with the patch.
   We add a set of useless codes like a bunch of NOP instructions which will be replaced when we write the actual patch. This also helps us, as we don't need to worry about calls and addresses as we are replacing useless code instead of adding a completely new patch. A bigger limitation is that we need access to source code which kinda
5. Analysis of the “LS” file.

In the analysis of the ls-files (version 1.1.2018 & 3.16.2018), we found that there are multiple patches with varying lengths and types of patches. This means that ls code files can be used as a real patch.

**Results**

As mentioned above, we worked with small binaries synthesized specifically for our use. Applying what we've learned on a real-world binary is a much more realistic and appealing exercise. Working with real-world binaries, however, is more nuanced and challenging.

We selected ls, a core Gnu utility on Linux systems for our real-world binary to work with. We began our work by locating in ls's git history a minimally sized patch so we could compare the effects on the binary of a nominal-sized patch, applied March 14th, 2108, with 15 additional lines, and 5 deletions from the base file ls.c. This relatively small patch resulted in a binary file size change of ~600 bytes.

Tracking the changes in pointers and other metadata for this small patch proved to be challenging. Unfortunately, we didn't have enough time to achieve a depth of knowledge sufficient to apply the patch to ls. As can be surmised, while we learned much from our synthesized binaries, we are still working on translating our new knowledge to real-world binaries. As of the date of this report, we are still working on the challenges of working with larger binaries.

**Conclusions and future work**

The promise of this work is in the lessons that we've learned that we can apply to work moving forward. Sometimes it's necessary to reconcile the requirements of a daunting challenge with our own preconceived ideas of what the task entails. In our case we found that the challenge was much more difficult and worthwhile than we originally thought, remaining unsolved.

Moving forward, the next step is to take time to learn the nuances of production real-world binaries and apply the lessons learned here to the more challenging world of binaries in the wild. This project and this class were very beneficial to us as a stepping stone in our path towards deeper knowledge of the software security field, of which we are grateful. Thank you for the opportunity to work on this project and for a great semester.