

# Erweiterung der Projektarbeit

Yusuf Enes  
Aalen University

Tamer  
Reverse Engineering

## Abstract

In der bisherigen Projektarbeit wurde das LS-Binary untersucht sowie auch modifiziert und letztlich im finalen Stadium auch rekonstruiert. In diesem Paper wird das "mv" nochmal genauer untersucht und dabei ist auch das Ziel, dieses anhand eines C++ Codes wieder zu rekonstruieren.

Zusätzlich wurde der C++ Code von dem touch Binary auch ausgelesen, sodass dieser direkt kompiliert werden kann.

## 1 Einleitung

Bei dem "mv"-Binary geht es eigentlich darum den Dateiorst einer anderen Datei zu ändern, weshalb es sich hier um einen Datentransfer handelt, sowie auch kann nur der Dateiname umbenannt werden und die Datei immer noch im selben Verzeichnis befinden.

## 2 C++-Code der Rekonstruktion

Dieser C++-Code konnte festgestellt werden anhand eines Strace-Tools. Dieser Code sieht folgendermaßen aus:

```
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int fd = openat(AT_FDCWD, "subtree", O_RDONLY
                   | O_PATH | O_DIRECTORY);
    renameat2(AT_FDCWD, "ls", fd, "ls2", RENAME_NOREPLACE);
    return 0;
}
```

Zuerst wird die `openat(..)` Funktion aufgerufen, das folgende Parameter bekommt.

1. AT\_FDCWD
2. "subtree"

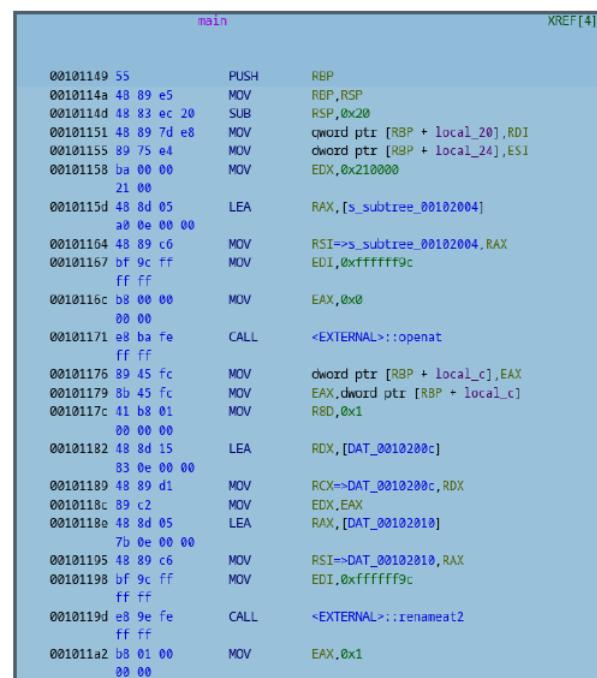
## 3. O\_RDONLY | O\_PATH | O\_DIRECTORY

Beim ersten Parameter handelt sich um ein Flag, das eine folgende Bedeutung hat: **Current-Work-Directory**.

Beim zweiten Parameter ist es eine String-Kette, welche den Namen des jeweiligen Zielverzeichnisses ausgibt.

Mit dem dritten Parameter wird damit der **Verzeichnis-**Deskriptor festgelegt und nicht ein File-Deskriptor. Der Verzeichnisdeskriptor ist der Rückgabewert von der `openat`-Funktion.

## 2.1 Assembly-Code



```
main XREF[4]

00101149 55      PUSH    RBP
0010114a 48 89 e5 MOV     RBP,RSP
0010114d 48 83 ec 20 SUB    RSP,0x20
00101151 48 89 7d e8 MOV     dword ptr [RBP + local_20],RDI
00101155 89 75 e4 MOV     dword ptr [RBP + local_24],ESI
00101158 ba 00 00 MOV     EDX,0x210000
                           21 00
0010115d 48 8d 05 LEA    RAX,[s_subtree_00102004]
                           a8 0e 00 00
00101164 48 89 c6 MOV    RSI=>s_subtree_00102004,RAX
00101167 bf 9c ff MOV    EDI,0xffffffff9c
                           ff ff
0010116c b8 00 00 MOV    EAX,0x0
                           00 00
00101171 e8 ba fe CALL   <EXTERNAL>::openat
                           ff ff
00101176 89 45 fc MOV    dword ptr [RBP + local_c],EAX
00101179 8b 45 fc MOV    EAX,dword ptr [RBP + local_c]
0010117c 41 b8 01 MOV    R8D,0x1
                           00 00 00
00101182 48 8d 15 LEA    RDX,[DAT_0010200c]
                           83 0e 00 00
00101189 48 89 d1 MOV    RCX=>DAT_0010200c,RDX
0010118c 89 c2 MOV    EDX,EAX
0010118e 48 8d 05 LEA    RAX,[DAT_00102010]
                           7b 0e 00 00
00101195 48 89 c6 MOV    RSI=>DAT_00102010,RAX
00101198 bf 9c ff MOV    EDI,0xffffffff9c
                           ff ff
0010119d e8 9e fe CALL   <EXTERNAL>::renameat2
                           ff ff
001011a2 b8 01 00 00 MOV    EAX,0x1
```

### 2.1.1 Beschreibung vom Assembly-Code

Beim Assembly-Code sind die SYSCALLS für die Funktionen **openat(..)** und **renameat2** zu sehen. Zuerst wird dafür der richtige Werte im Stack initialisiert. Daraufhin wird das RAX mit der Adresse von der String-Kette "subtree" geladen und diese dann in das RSI-Register gespeichert. Im EDX-Register befindet sich das Current-Working-Directory-Flag mit der kurzen Bezeichnung **AT\_FDCWD** und dem Hex-Wert **0x210000**. Anderseits enthält das EDI-Register den Wert **0xfffff9c**, was mit einer ODER-MASKIERUNG von den 3-FLAGS berechnet wurde.

### 2.1.2 Rekonstruierung der C-Funktion rename2

Grundsätzlich besteht diese C-Funktion aus mehreren Funktionen. In diesem Fall muss diese Funktion auch auf das Dateisystem zugreifen mit Funktionen wie **opendir**, **dirfd**, **readdir**, **openat**, und **stat**. Diese Funktionsaufrufe von diesen Programmen kennzeichnen die Grundfunktionalität des ls-Binary, und somit hier ein Einleseverfahren notwendig wird, um die Dateideskriptoren zu bekommen. Nachdem die Deskriptoren ausgelesen worden sind, können diese übergeben werden. Dies geschieht dadurch, in dem sie an eine Stackadresse gemappt werden. Diese Funktion sieht folgendermaßen aus.

```
void * addr = mmap(NULL, statbuf.size, PROT_READ | PROT_WRITE,  
MAP_SHARED | MAP_FILE, gd, 0);  
  
void * addr2 = mmap(NULL, statbuf.st_size, PROT_READ | PROT_WRITE,  
MAP_SHARED | MAP_FILE, hd, 0);
```

1. Quelladresse addr
2. Zieladresse addr2

Nach dieser Funktion werden, die folglichen Funktionen verwendet:

```
ftruncate(fd, statbuf.st_size);  
memmove(addr2, addr, statbuf.st_size);
```

Hierbei erfolgt ein Datentransfer, mit der, eine Datenübertragung zwischen zwei Stackadressen erfolgt. Da diese Stackadressen auf das Dateisystem gemappt worden sind, findet auch gleichzeitig eine Datenübertragung im Dateisystem statt. Dieser vorliegender Code, soll die internen Funktionen des **rename2** Funktionsaufrufes zeigen. Damit ist das ein SYSCALL was eine mehrdimensionale Funktionalität aufweist.

## 3 touch-Binary

Dieses Binary ist dazu da, um Dateien zu erstellen, das auf der Festplatte dann liegt, welche mit dem Dateisystem **EXT4** bereits formatiert wurde.

```
#include <stdio.h>  
#include <unistd.h>  
int main(int argc, char * argv[]){  
    FILE * a = fopen(argv[1], "w");  
    int x = fileno(a);  
    lseek(x, 1, 0);  
    close(x);  
    return 1;  
}
```

### 3.1 SYSCALL-FILENO

Der folgende SYSCALL-FILENO sieht folgendermaßen aus:

0010dce1 89 54 24 0c	MOV	dword ptr [RSP+c], EDX
0010dce5 48 89 34 24	MOV	qword ptr [RSP], RSI
0010dce9 e8 12 56 ff ff	CALL	<EXTERNAL>::fileno
0010dcee 8b 54 24 0c	MOV	EDX,dword ptr [RSP+c]
0010dcf2 48 8b 34 24	MOV	RSI,qword ptr [RSP]=>b
0010dcf6 89 c7	MOV	EDI,EAX
0010dcf8 e8 f3 54 ff ff	CALL	<EXTERNAL>::lseek