CVE-2023-38831 분석 보고서

투모로우(to-more-row) / 공개된 1-day 취약점 상세 분석팀

01. 요약

1) CVE-2023-38831 개요

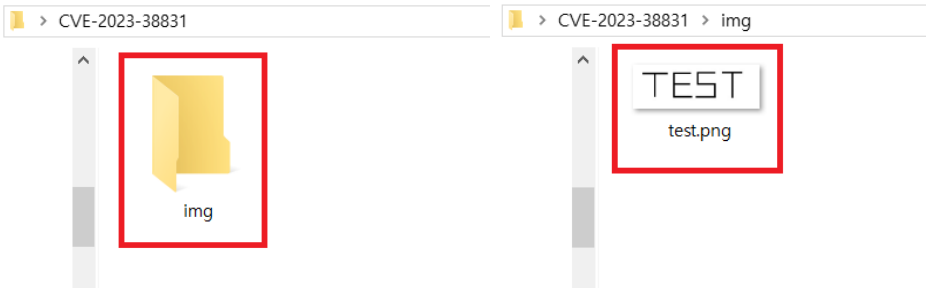
WinRAR의 6.23 이하 버전에서 발견된 CVE-2023-38831 취약점은 파일 이름에 대한 공백 처리 로직에서 발생하는 문제로, 이로 인해 원격 코드 실행(Remote Code Execution, RCE)이 가능한 상태가 된다.

02. 사전지식

1) 재현 환경 구성

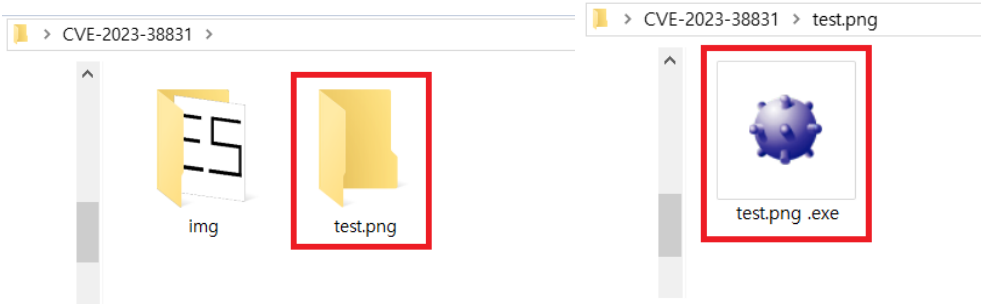
1. WinRAR 6.22 다운로드

2. 취약점 환경 구성



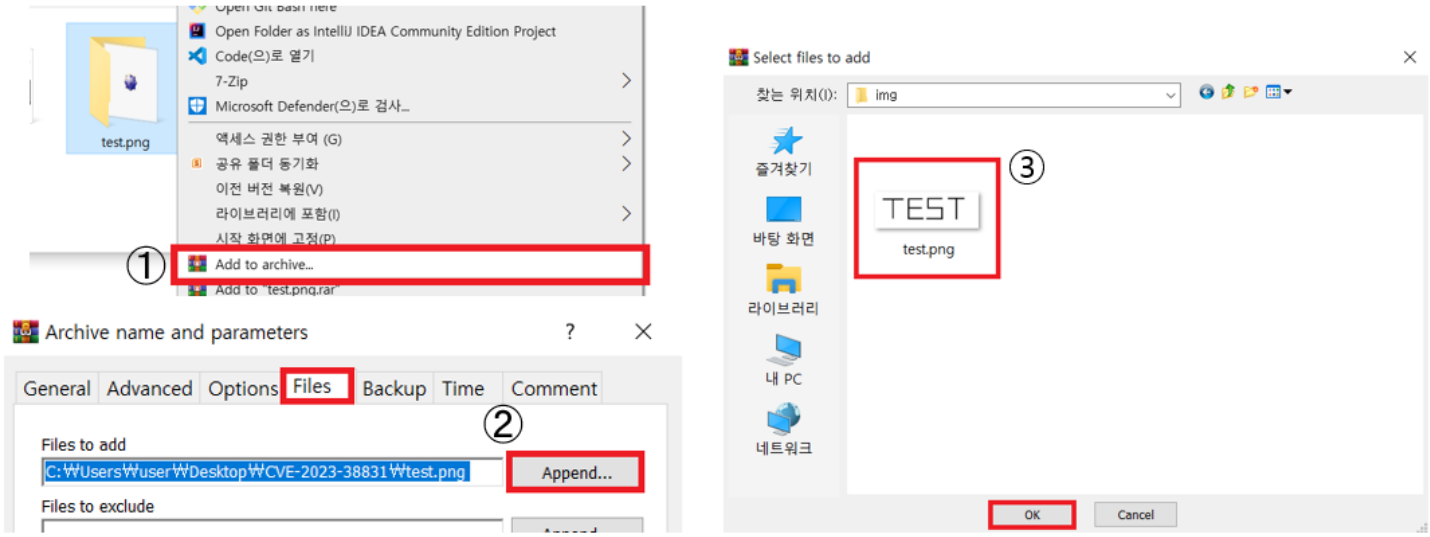
[그림 1] 취약점 환경 구성 - 1

“img” 폴더를 생성하고, 해당 폴더에 “test.png” 이미지 파일을 넣는다.



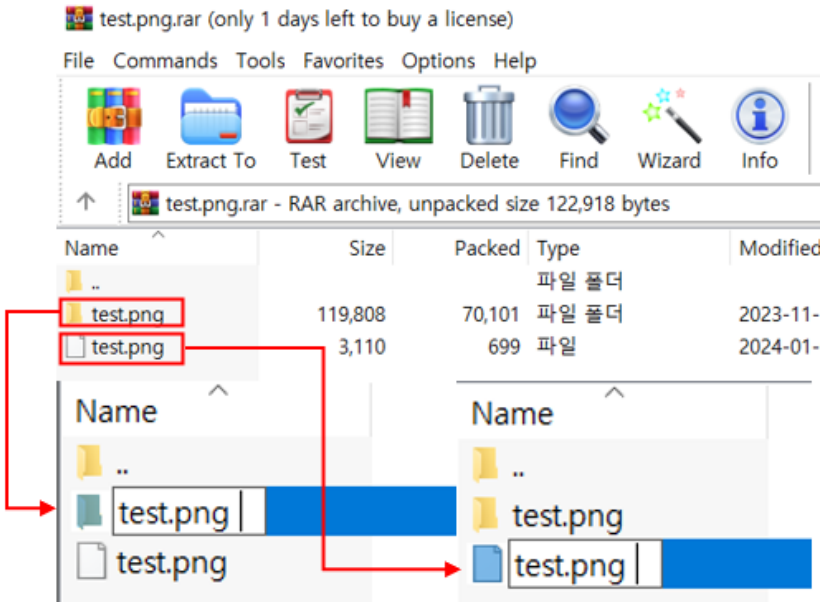
[그림 2] 취약점 환경 구성 - 2

이미지 파일과 동일한 이름의 “test.png” 폴더를 생성하고, “test.png” 폴더에 임의 실행 파일을 추가한다.
이름은 “test.png .exe“ 으로 지정한다.



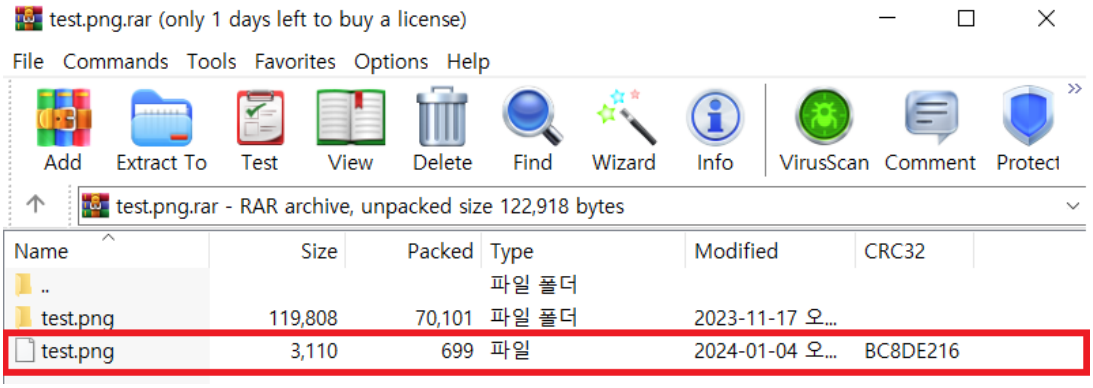
[그림 3] 취약점 환경 구성 - 3

“test.png” 폴더를 우클릭하여 WinRAR의 Add to archive 항목을 선택하고, Files 탭에서 Append 클릭한 후 “test.png” 이미지 파일을 선택한다.



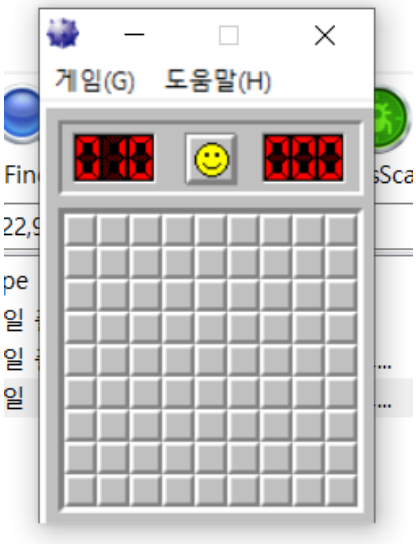
[그림 4] 취약점 환경 구성 - 4

파일과 폴더 이름에 각각 공백을 추가한다.



[그림 5] 취약점 환경 구성 - 5

그 후, “test.png “ 파일을 실행시킨다.



[그림 6] 취약점 환경 구성 - 6

의도와 다르게, “test.png \test.png .exe” 가 실행된다.

2) 디버깅 도구

x64dbg : 64비트 플랫폼에서 동작하는 오픈 소스 디버깅 도구로, 코드 분석, 메모리 검사, 어셈블리어 레벨의 디버깅을 지원하는 소프트웨어 역공학 도구이다.

IDA : 디컴파일러 및 디버거로, 다양한 플랫폼 지원, 그래픽 사용자 인터페이스, 자동 분석 및 기호화, 플러그인 지원을 특징으로 하는 소프트웨어 역공학 도구이다.

Process Monitor : Windows 운영 체제에서 실행되는 프로세스의 행동을 실시간으로 모니터링하고 기록하는 도구이다.

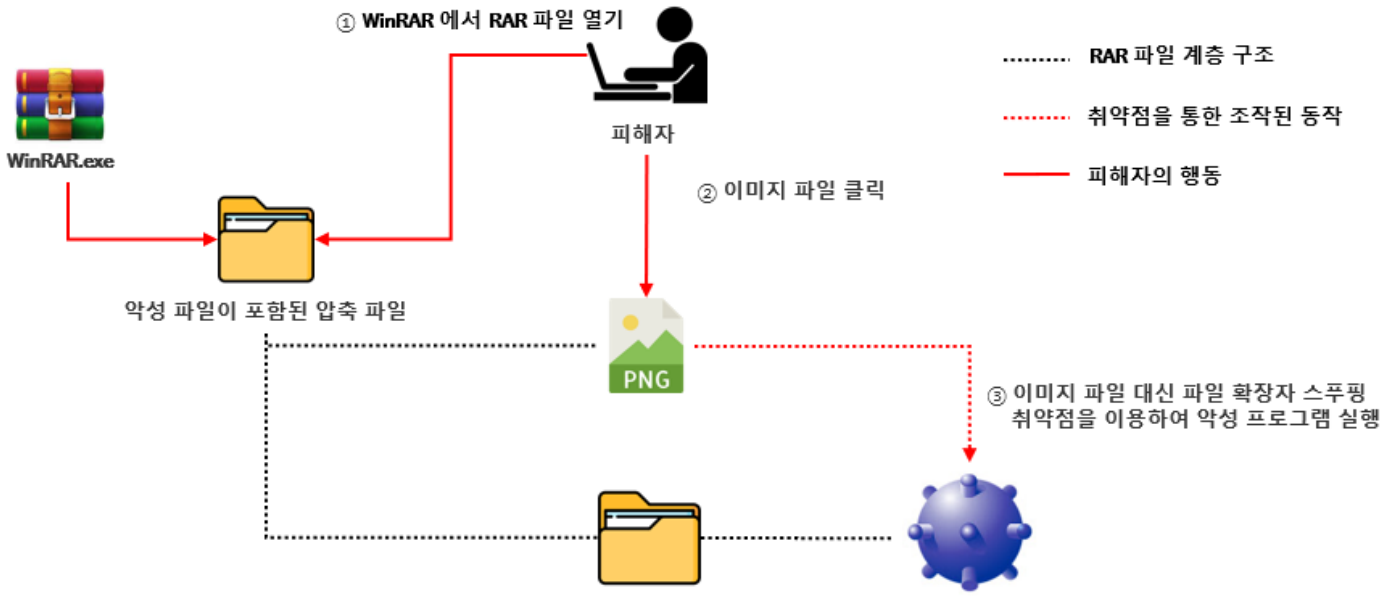
BinDiff : 이진 파일 간의 차이를 비교하는 도구로, 주로 소프트웨어 역공학에서 사용되며 함수 수준에서의 변경을 시각화하여 코드 분석과 보안 취약점 탐지를 수행하는 도구이다.

03. 상세 분석

a) 취약점 발생 원리

CVE-2023-38831 취약점은 WinRAR의 잘못된 파일 이름에 대한 공백 처리 로직으로 인해 발생한다.

아래 [그림 7]은 CVE-2023-38831 취약점이 발생되는 과정을 정리한 그림이다.



[그림 7] CVE-2023-38831을 활용하는 감염 체인

b) 파일 비교와 생성

```
while ( Read_FileHeader(v26) )          // File Block의 헤더를 읽기
{
  sub_1400ACB0C();
  if ( v27 == 5 )
  {
    v13 = v35;
    break;
  }
  if ( v27 == 2 )
  {
    if ( (unsigned int)(v37 - 1) <= 1 && v29 <= 0xF )
      break;
    if ( !v30 )
    {
      if ( !v11 && !v32 )
      {
        if ( !v14 )
          sub_1400ABFF4(*(_QWORD *)(a1 + 32), v39, 2048i64);
        if ( !v15 )
          *(_QWORD *)(*(_QWORD *)(a1 + 32) + 4096i64) = v36;
      }
      if ( (unsigned int)Compare_ClickedFile(*(_QWORD *)(a1 + 56), (unsigned int)&v28, 0, 6, 0, 0i64, 0) )// File Block 헤더에 담긴 파일 이름과 더블 클릭한 파일 이름을 비교
```

Name	Value	Start	Size	Type
Signature[8]		0h	8h	ubyte
Block[0]	Main block	8h	11h	struct RarBlockV5
Block[1]	File block: test.png /test.png .exe	19h	11204h	struct RarBlockV5
Block[2]	File block: test.png	1121Dh	DF8h	struct RarBlockV5
Block[3]	File block: test.png /	12018h	28h	struct RarBlockV5
Block[4]	Service (Quick open) block	12040h	57h	struct RarBlockV5
Block[5]	End block	12097h	8h	struct RarBlockV5

[그림 8] 파일 비교 - 1

파일 블록의 헤더를 읽고, 파일 이름들을 추출한 후 클릭한 파일의 이름과 비교한다.

```
char __fastcall Compare_File(WCHAR *Clicked_File, WCHAR *Second_File, unsigned int Flag)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v3 = (unsigned __int16)Flag;
  v4 = Flag >> 31;
  if ( (_WORD)Flag )
  {
    v7 = -1i64;
    v8 = -1i64;
    do
      ++v8;
    while ( Clicked_File[v8] );
    if ( (unsigned int)(unsigned __int16)Flag - 2 > 2 )
    {
      if ( (Flag & 0x80000000) == 0 )
        v9 = CompareN_FileName(Clicked_File, Second_File);
      else
        v9 = wcsncmp(Clicked_File, Second_File, v8);// Clicked_file 길이만큼 비교
      if ( !v9 )
      {
        v10 = Second_File[v8];
        if ( v10 == 92 || v10 == 47 || !v10 )
          return 1;                         // 파일이나 폴더로써 이름이 같다면 1 반환
      }
      if ( v3 == 1 )
        return 0;
    }
    CutString(Clicked_File, (wchar_t *)String1);// '\' 또는 '/' 다음 문자 거리만큼 문자열 복사
    CutString(Second_File, String2);            // '\' 또는 '/' 다음 문자 거리만큼 문자열 복사
    if ( ((v3 - 2) & 0xFFFFFFFD) == 0 )
    {
      if ( (_BYTE)v4 )
      {
        v13 = String1;
        v14 = (char *)String2 - (char *)String1;
        while ( 1 )
        {
          v15 = *v13;                       // String1에서 현재 가리키고 있는 문자를 가져옴
          if ( *v13 != *(__int16 *)((char *)v13 + v14) )// 두 문자열의 같은 위치 비교, 다르면 반복문 종료
            break;
          ++v13;
          if ( !v15 )
          {
            v16 = 0;
            goto LABEL_22;
          }
        }
        v16 = v15 < *(unsigned __int16 *)((char *)v13 + v14) ? -1 : 1;
      }
      else
      {
        v16 = Compare_FileName((PCNZWCH)String1, String2);// 다르면 1, 같으면 0 반환
      }
```

0x271288FCCF0i64:L"test.png "

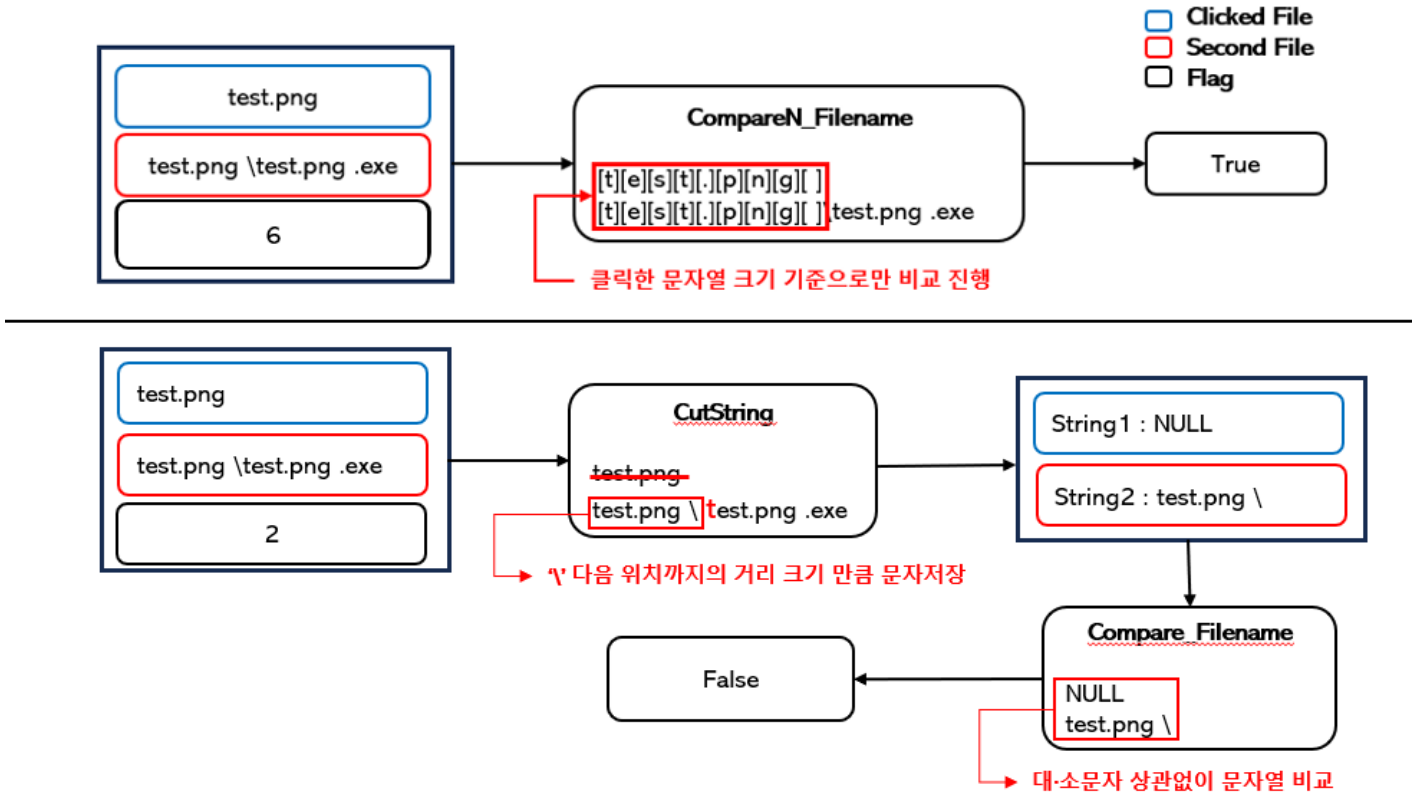
0x82708DD488i64:L"test.png ₩₩t…

Flag : 6

Flag : 2

[그림 9] 파일 비교 - 2

파일 이름의 길이는 클릭한 파일의 문자열 크기를 기준으로 계산된다. 취약점의 근본적인 원인은 Flag가 6으로 설정되어 잘못된 비교 결과로 인해 악성 파일까지 임시 폴더에 생성된다. Flag 값에 따라 비교되는 과정은 [그림 10]과 같다.



[그림 10] Flag 값에 대한 파일 비교 과정

CompareN_Filename 함수를 호출하여 두 문자열을 비교할 때, 클릭한 파일의 이름 문자열 크기 기준으로만 비교를 하기 때문에 뒤에 문자열이 오더라도 비교하지 않는다.
악성 파일의 경우, 클릭한 파일의 이름 문자열 크기 만큼 비교했을 때 일치하기 때문에 결과적으로 True를 반환하게 된다.
Flag 값에 따라 생성되는 파일 결과는 [그림 11]과 같다.

Flag 값 / 생성된 파일	2	6
악성 파일	X	O
클릭한 파일	O	O

[그림 11] Flag 값에 따라 생성된 파일

```
do
{
  FileHeader = Read_FileHeader(v31);          // 파일 블록 헤더 정보 가져오기
  if ( FileHeader )
  {
    v20 = *(_QWORD *)(a1 + 56);
    if ( *(_BYTE *)(v20 + 29457) )
    {
      if ( *(_WORD *)(v20 + 54338) != 73 && v34 == 3 && v36 == 82 && v37 == 82 && !v38 )
      {
        v18 = 1;
        if ( v32 != 1 )
          sub_14009FBD8(v31);
      }
    }
  }
  v25[0] = 0;
}
while ( (unsigned __int8)CreateFile(a1, v31, FileHeader, v25) );// 비교한 결과를 토대로 파일 생성
```

Block[0]	Main block
Block[1]	File block: test.png /test.png .exe
Block[2]	File block: test.png
Block[3]	File block: test.png /

[그림 12] 파일 생성 - 1

파일 블록에 대한 비교 결과를 토대로 파일이 생성되며, 때문에 CreateFile 함수를 호출하여 클릭했던 파일과 악성 파일이 생성된다.

```
void __fastcall SpaceDelete(_WORD *FileName, unsigned __int64 a2)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v18 = a2;
  Length = 0i64;
  v19 = FileName;
  for ( i = FileName; ; ++i )
  {
    CurrentChar = *i;                         // 현재 문자를 할당
    if ( *i == '\\' || CurrentChar == '/' || !CurrentChar )// 현재 문자가 '\' 이거나 '/' 또는 NULL 일때
    {
      for ( j = Length - 1; j > 0; --j )      // 끝에서부터 문자를 거꾸로 읽어옴
      {
        if ( FileName[j] != ' ' )             // 문자가 공백이 아닐 때
        {
          if ( FileName[j] != '.' )           // '.' 이 아니면 반복문 종료
            break;
          v7 = FileName[j - 1];
          if ( v7 == '\\' || v7 == '/' || v7 == '.' && j == 1 )// 문자가 '\', '/', '.' 이거나 1이면 반복문 종료
            break;
        }
        --Length;
      }
    }
    FileName[Length] = CurrentChar;           // 공백을 제거하고 해당 자리에 NULL 저장
    if ( !*i )
      break;
    ++Length;
  }
```

[그림 13] 파일 생성 - 2

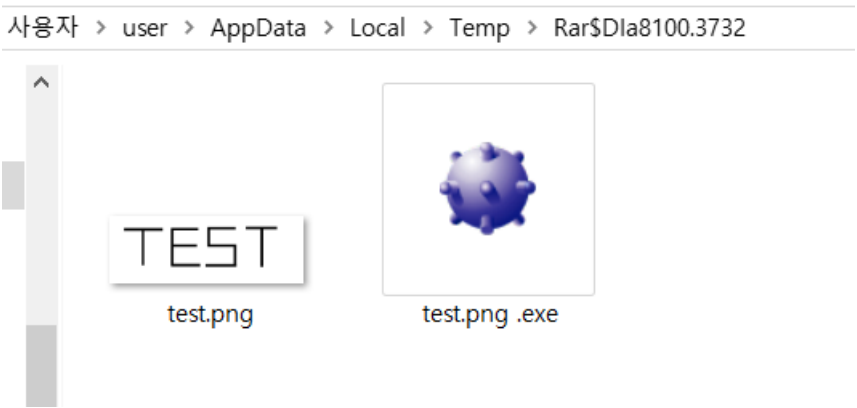
Windows에서 파일 이름의 양쪽 끝 부분에 공백이 존재하면 안되므로, 파일 생성 전 SpaceDelete 함수를 호출하여 클릭한 파일 이름에 대한 공백을 제거한다.

c) 압축 해제

```
result = TempDir(L"Rar$DI", Tempdir_path, 2048i64, a4);// 임시 폴더 생성, 해당 경로 저장
if ( result )
{
  LOBYTE(v8) = 1;
  sub_1400E2CDC(v8);
  sub_1400097E8(&unk_1401BA630, &szShortPath, 2048i64);
  dword_1401A4928 = dword_1401BA634;
  sub_1400ABFF4(&word_14019D54E, Tempdir_path, 2048i64);
  dword_1401A3914 = 2;
  dword_1401A259C = 1;
  sub_1400AC058(&unk_1401AA998, FileBlock);
  Unzip_Temp(&unk_1401BA630, 0i64, 0i64);     // 압축 해제 루틴
  v9 = 0;
  FileName = (wchar_t *)Unzip_Path(FileBlock);// FileBlock에서 클릭한 파일명 추출
  if ( !(unsigned __int8)sub_140095FD0(FileBlock) )
  {
    FilePath(Tempdir_path, FileName, SaveLocation, StringLength);// 임시 폴더 경로에 파일명을 더해서 저장
    if ( !*FileName )
    {
ABEL_20:
      Remove_TempDir(Tempdir_path);
      goto LABEL_21;
    }
    if ( (unsigned __int8)Remove_dir(SaveLocation) )
    {
```

```
43 00 3A 00 5C 00 55 00 73 00 65 00 72 00 73 00  C.:.\.U.s.e.r.s.
5C 00 6D 00 63 00 6B 00 6B 00 6B 00 5C 00 41 00  \.m.c.k.k.k.\.A.
70 00 70 00 44 00 61 00 74 00 61 00 5C 00 4C 00  p.p.D.a.t.a.\.L.
6F 00 63 00 61 00 6C 00 5C 00 54 00 65 00 6D 00  o.c.a.l.\.T.e.m.
70 00 5C 00 52 00 61 00 72 00 24 00 44 00 49 00  p.\.R.a.r.$.D.I.
61 00 36 00 38 00 31 00 32 00 2E 00 33 00 33 00  a.6.8.1.2...3.3.
30 00 38 00 35 00 5C 00 74 00 65 00 73 00 74 00  0.8.5.\.t.e.s.t.
2E 00 70 00 6E 00 67 00 20 00 00 00 00 00 00 00  ..p.n.g. .......
```

[그림 14] 압축 해제 - 1



[그림 15] 압축 해제 - 2

TempDir 함수를 호출하여 “Rar$DI”로 시작하는 임시 폴더를 생성하고 경로를 저장하며, 해당 경로는 압축 해제 시 파일이 저장되는 위치이다. 이 후 Unzip_Temp 함수를 호출하여 압축 해제를 진행하고, FileName 변수에 파일 이름을 FileBlock에서 추출하여 저장한다. 때문에, FileName에 “test.png “로 공백이 포함된 파일 이름이 들어간다. FilePath 함수를 호출하여 임시 폴더 경로와 파일 이름(“test.png “)을 더하여 파일 경로를 저장한다. 따라서 결국 파일 경로 뒤에 공백이 붙게 된다. 결과적으로 사용자가 클릭한 파일과, 악성 파일이 압축 해제 된다.

d) 파일 실행

```
SHELLEXECUTEINFOW pExecInfo; // [rsp+20h] [rbp-E0h] BYREF
char Buffer[8192]; // [rsp+90h] [rbp-70h] BYREF

sub_14011C300(&pExecInfo, 0i64, 112i64);
v10 = 1280;
pExecInfo.cbSize = 112;
pExecInfo.lpFile = fileName;
pExecInfo.lpVerb = a7;
if ( !a5 )
  v10 = 1344;
pExecInfo.fMask = v10;
v11 = (_WORD *)sub_140095928(fileName);
if ( !v11 || *v11 == 46 && !v11[1] )
{
  pExecInfo.fMask |= 1u;
  pExecInfo.lpClass = L".";
}
pExecInfo.lpDirectory = directory;
pExecInfo.lpParameters = a4;
if ( (const WCHAR *)sub_140096754(fileName) == fileName && !(unsigned __int8)sub_140094B74(fileName, L"exe") )
{
  sprintf_s(Buffer, 0x1000ui64, L".\\%s", fileName, *(_QWORD *)&pExecInfo.cbSize);
  pExecInfo.lpFile = (LPCWSTR)Buffer;
}
pExecInfo.nShow = 1;
byte_14018A805 = 1;
v12 = ShellExecuteExW(&pExecInfo);
```

[그림 16] execute_file 함수

이후 ShellExecuteExW API를 호출한다. ShellExecuteExW 함수는 전달받은 파일 이름과 임시 폴더 경로를 PathFindExtensionW 함수에 전달하여 임시 폴더에서 실행할 파일을 찾는다. PathFindExtensionW 함수는 주어진 경로에서 전달받은 파일 이름과, 파일의 확장자를 확인하여 해당 파일을 찾아내는 함수이다.

```
unsigned __int64 __fastcall kernelbase_PathFindExtensionW(unsigned __int64 a1)
{
  unsigned __int16 *startOfExtensionPointer; // r8
  unsigned __int64 endOfExtension; // rdx
  unsigned __int16 *currentCharPointer; // rax
  unsigned __int16 currentChar; // cx
```

[그림 17] PathFindExtensionW 함수 - 1

*startOfExtensionPointer는 파일 확장자의 시작 지점을 가리키는 포인터이며, *currentCharPointer는 현재 문자를 가리키는 포인터이다. currentChar 변수는 현재 문자 값을 담는 변수로 사용된다.

```
while ( 1 )
{
  currentChar = *currentCharPointer;
  if ( !*currentCharPointer )
    break;
  if ( currentChar <= 0x5Cu )
  {
    if ( currentChar == '.' )
    {
      startOfExtensionPointer = currentCharPointer;
    }
    else if ( currentChar == ' ' || currentChar == '\\' )
    {
      startOfExtensionPointer = 0i64;         // null
    }
  }
  if ( ++currentCharPointer >= endOfExtension )
    return &a1[(int)((__int64 (__fastcall *)(unsigned __int16 *, unsigned __int16 *, unsigned __int16 *))kernelbase_lstrlenW)(
                       a1,
                       endOfExtension,
                       startOfExtensionPointer)];
}
```

[그림 18] PathFindExtensionW 함수 - 2

currentCharPointer를 증가시키며 현재 문자 값을 currentChar에 저장하고, currentChar가 '.'이면 startOfExtensionPointer에 현재 위치 포인터 currentCharPointer를 할당한다. 왜냐하면 대부분의 확장자는 '.' 뒤에 있기 때문이다.

```
while ( 1 )
{
  currentChar = *currentCharPointer;
  if ( !*currentCharPointer )
    break;
  if ( currentChar <= 0x5Cu )
  {
    if ( currentChar == '.' )
    {
      startOfExtensionPointer = currentCharPointer;
    }
    else if ( currentChar == ' ' || currentChar == '\\' )
    {
      startOfExtensionPointer = 0i64;         // null
    }
  }
  if ( ++currentCharPointer >= endOfExtension )
    return &a1[(int)((__int64 (__fastcall *)(unsigned __int16 *, unsigned __int16 *, unsigned __int16 *))kernelbase_lstrlenW)(
                       a1,
                       endOfExtension,
                       startOfExtensionPointer)];
}
```

[그림 19] PathFindExtensionW 함수 - 3

하지만 "test.png " 는 확장자 뒤에 공백이 있기 때문에, 마지막에 startOfExtensionPointer에 null 값으로 초기화하고 루프를 빠져나간다.

```
if ( startOfExtensionPointer )
  return startOfExtensionPointer;
```

[그림 20] PathFindExtensionW 함수 - 4

반복문을 빠져나오면서 if 문을 거치는데, startOfExtensionPointer의 값이 null이므로 Invalid 처리되지만, 다음 "test.png .exe"에 대한 PathFindExtensionW 함수에서는 확장자 뒤에 공백이 없기 때문에 정상적으로 startOfExtensionPointer의 값이 반환되어 "test.png .exe"가 실행시킬 파일로 지정된다.

[그림 21] StrCmpIW 함수

마지막으로 ShellExecuteExW 함수는 StrCmpIW 함수를 통해 파일의 실행 우선순위를 정한다. 우선순위가 가장 높은 파일이 실행되며, 우선순위는 파일의 확장자를 통해 정해진다.

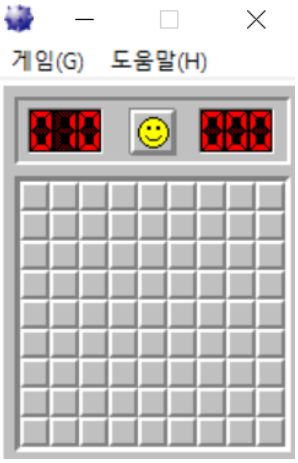
```
L".exe"
L".pif"

L".exe"
L".com"

L".exe"
L".exe"
```

[그림 22] StrCmpIW 함수의 비교 순서

우선 순위를 검사하는 확장자 순서는 '.pif', '.com', '.exe', '.bat', '.lnk', '.cmd' 순으로 진행된다. 하지만 "test.png .exe"의 확장자는 '.exe' 이기 때문에 '.pif', '.com', '.exe' 까지만 문자열 비교를 하게 된다.

[그림 23] Jump If Equals

"test.png .exe" 파일의 확장자와 '.exe'를 비교할 때, 확장자가 동일하기 때문에, 이 je 어셈블리어를 통해 파일을 실행하는 코드로 이동한다. 결론적으로 "test.png .exe" 파일이 실행되는 것이다.



[그림 24] “test.png .exe”이 실행됨

따라서 ShellExecuteExW 함수가 “test.png .exe” 파일을 실행시킨다.

e) 패치 내역

1) 삭제

[그림 25] BinDiff를 활용한 패치 내용 비교 (좌:패치이전 / 우:패치이후)

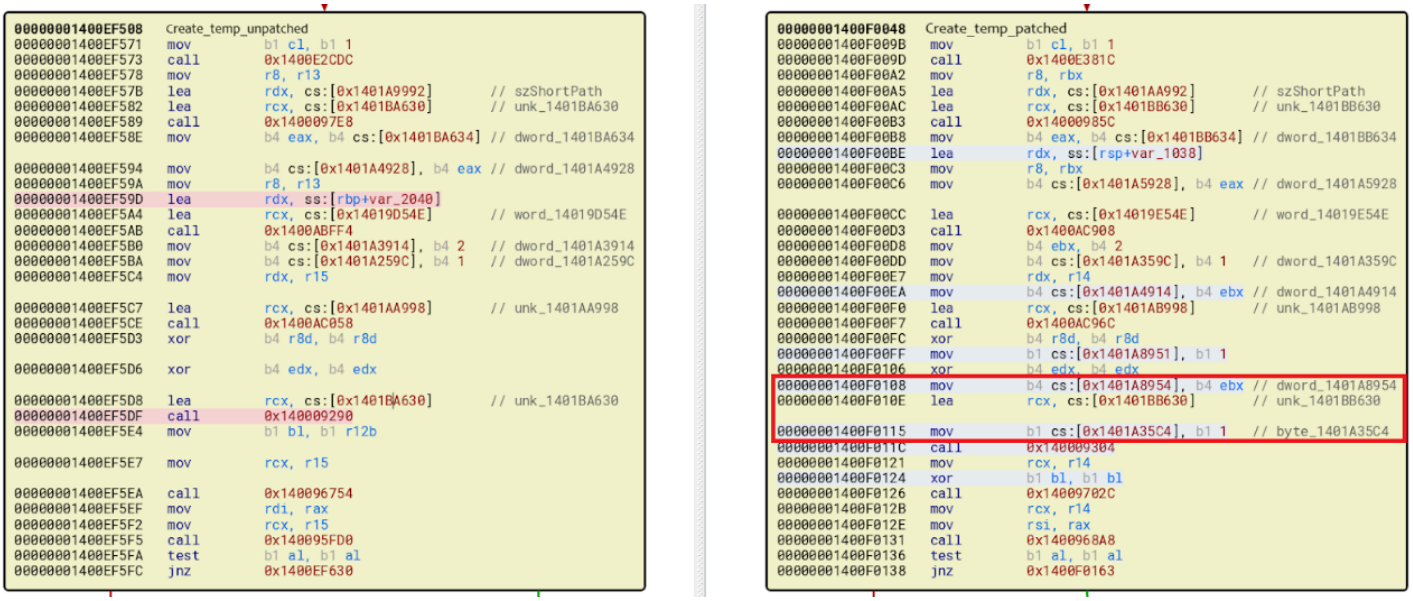
패치 이전의 버전(6.22)과 패치 이후의 버전(6.24)에서 유사도는 64%로 높지않은 퍼센트를 보이고 있다.
[그림 22]에서 보이는 것 처럼 함수가 전부 삭제 되었기 때문이다. 하지만 삭제 되었다고 해당 부분에 존재한 함수가 제거된 것은 아니다.

Similarit	EA Primary	Name Primary
1.00	0000000140096014	sub_140096014
1.00	0000000140095F50	sub_140095F50
1.00	00000001400868B0	sub_1400868B0
1.00	0000000140086A38	sub_140086A38
1.00	0000000140086960	sub_140086960
1.00	00000001400ABFF4	sub_1400ABFF4
1.00	0000000140096754	sub_140096754
1.00	00000001400AF078	sub_1400AF078
1.00	00000001400868C8	sub_1400868C8

[그림 26] BinDiff와 IDA를 활용하여 유사도 파악

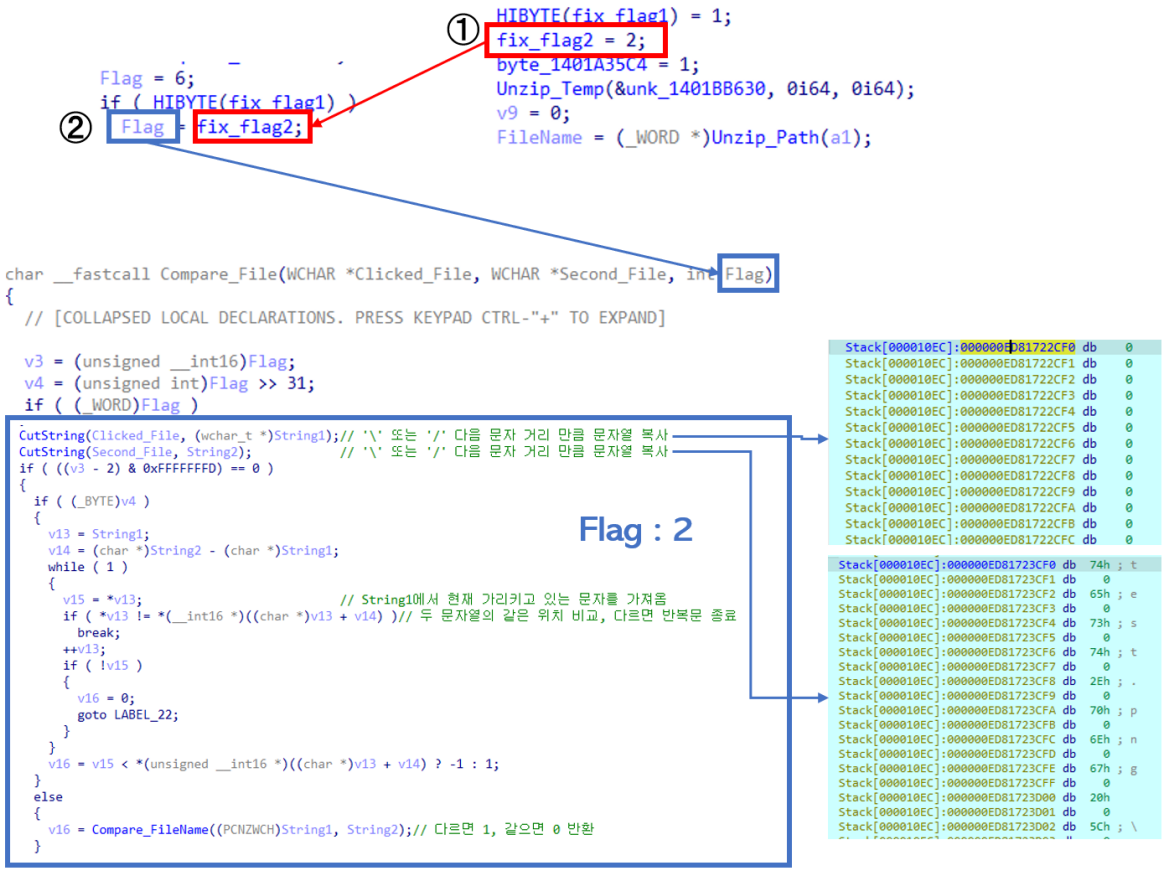
[그림 25] 에서 삭제된 함수에 대한 유사도를 확인할 수 있다. 검사 결과 유사도가 1.00으로 다른 함수로 옮겨져 해당 함수들이 사용되고 있음을 알 수 있다.

2) 추가



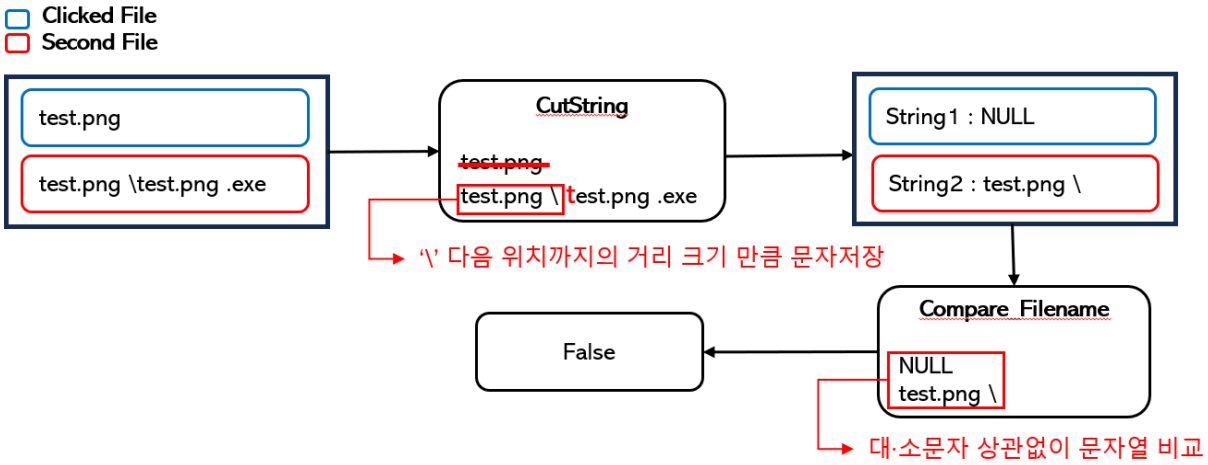
[그림 27] BinDiff를 활용한 추가된 내용 비교 (좌:패치 이전 / 우:패치 이후)

취약한 버전과 패치된 버전의 차이점을 확인했을 때 Flag 값을 추가하고 있음을 [그림 27]에서 확인할 수 있다.



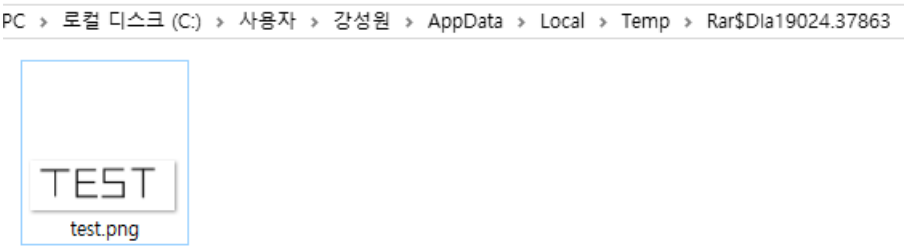
[그림 28] Flag 값이 바뀌는 과정과 실행되는 함수

[그림 28]에 정의된 순서대로 진행되며, 기존에 설정되었던 Flag 값이 6에서 2로 수정되었다.
클릭한 파일과 악성 파일에 대한 문자열 비교 과정은 [그림 29]과 같다.



[그림 29] Flag가 2일 때 문자열 비교 과정

CutString 함수를 호출하여 '\' 또는 '/' 다음 문자 위치 거리 만큼 문자열을 복사하고 저장하고 Compare_Filename 함수를 호출하여 대·소문자 상관없이 문자열을 비교하여 False를 반환한다. 따라서 악성 파일 블록 헤더의 비교 결과가 일치하지 않게 되며, 이로 인해 악성 파일의 압축 해제도 진행되지 않게 된다.



[그림 30] 압축 해제 결과

압축 해제 결과 [그림 30]과 같이 클릭한 파일만 임시 폴더에 압축 해제가 되며 악성 파일이 아닌 클릭한 파일이 열리게 된다.
해당 취약점은 Flag 값에 따라 실행되는 파일 이름에 대한 문자열 비교 함수에서 문제가 발생하였고 Flag 값을 추가하여 문자열을 옳바르게 비교하도록 패치하였다.

05. 참고 문헌

[1]
https://googleprojectzero.github.io/0days-in-the-wild//0day-RCAs/2023/CVE-2023-38831.html, googleprojectzero

[2]
https://www.trellix.com/about/newsroom/stories/research/cve-2023-38831-navigating-the-threat-landscape-of-the-latest-security-vulnerability/, trellix, Neeraj Kumar Singh

[3]
https://www.mcafee.com/blogs/other-blogs/mcafee-labs/exploring-winrar-vulnerability-cve-2023-38831/, mcafee, Neil Tyagi

[4]
https://b1tg.github.io/post/cve-2023-38831-winrar-analysis/, b1tg