tremwil / ds3-nrssr-rce Public

Documentation and proof of concept code for CVE-2022-24125 and CVE-2022-24126.

123 stars 7 forks

Star Notifications

Code Issues Pull requests Actions Projects Security Insights

main Go to file

tremwil Update README.md ... on Aug 29 15

View code

README.md

Update: Dark Souls III 1.15.1

A new game update, 1.15.1, has been released for Dark Souls III on 2022/08/25, along with the restoration of online services. **This update fixed both CVE-2022-24125 and CVE-2022-24126**, along with a wide variety of other potential security vulnerabilities present in the game's P2P networking (OOB reads/writes). Furthermore, all known exploits allowing one to corrupt the save of other players have been fixed. Many common petty cheats (e.g. "curse knife") which could be encountered often during online multiplayer have also been patched.

ds3-nrssr-rce

This repository contains proof of concept code and documentation for the most recent RCE exploit affecting FROM SOFTWARE games, CVE-2022-24126. While theoretically possible in other games, focus is on Dark Souls III as this is the game my research has been conducted on. As of now proof of concept code only exists for Dark Souls III, the vulnerability has been confirmed to be present in:

- Dark Souls 1 PTDE (credit: LukeYui)
- Dark Souls Remastered (credit: metal-crow)

- Dark Souls 2 (including Scholar) (credit: LukeYui)
- Dark Souls 3 (up to 1.15.0) (credit: tremwil)

The vulnerable code is also present in Sekiro (credit: LukeYui), although there is no way to trigger it. Presence in Demon's Souls has not been confirmed but is very likely. **While the closed network test was affected by this, the release version of Elden Ring is not.** In fact, a huge list of network crashes, out-of-bounds reads/writes and exploits allowing players to modify the game data of peers which were present in Dark Souls III have been patched in Elden Ring. Kudos to LukeYui for compiling this list and to FROM SOFTWARE for acting swiftly! I'm happy to say that **Elden Ring is undisputably the safest FROM SOFTWARE title when it comes to the extent of the damage hackers can inflict.**

Dispelling Misconceptions

Contrary to popular belief, this is NOT a peer-to-peer networking exploit. It is related to the matchmaking server and thus much more severe, since you do not need to partake in any multiplayer activity to be vulnerable due to another matchmaking server vulnerability (CVE-2022-24125).

In Dark Souls III, A malicious attacker abusing this would have been able to reliably execute a payload of up to 1.3MiB[1] of shellcode on every online player's machine within seconds.

With the game having an average concurrent playerbase of about 20,000 players in the months preceding the server shutdown, it was clearly an issue that needed fixing immediately, especially with the possibility of it being in Elden Ring. Since FROM SOFTWARE had not yet acted over 40 days after my initial report with proof of concept videos and detailed exploit documentation (which a large part of this readme is based on), I decided to demonstrate the existence of the exploit puclicly in a benign manner in the hopes of raising attention to have it addressed by the developers, and it worked.

Table of Contents

- Exploit Summary (CVE-2022-24126)
- Distribtution Vectors (CVE-2022-24125)
- The General Exploitation Tactic for All Games
    - Bug #1: No Bounds Check in Entry List Parser
    - Bug #2: Buffer Overrun in `NRSessionSearchResult` Parser
    - Paving the way for the ROP chain

- Dark Souls III Proof of Concept Code
  - Running the PoC code
  - Attack Vector
  - Virtual Call Redirection Chain
  - Extra Information

Exploit Summary (CVE-2022-24126)

Improper bounds checking on a stack buffer and data size field during the parsing of `NRSessionSearchResult` matchmaking data allows an attacker to execute arbitrary code. The stack overflow allows one to overwrite the lower two bytes `vftable_ptr` of the `DLMemoryInputStream` object used internally by the stream reader, redirecting execution to carefully chosen neighboring code. Clever exploitation of the `DLMemoryInputStream` object's structure and data size field then allows one to achieve arbitrary code redirection, with `RCX` pointing to the address of our packet. From there a series of code redirections via virtual calls with different offsets (which will now jump at whatever addresses we wrote into the packet buffer) can be used to achieve arbitrary code execution.

Distribution vectors (CVE-2022-24125)

The distribution vectors are what make this particular RCE particularly serious (beyond already being an RCE). The exploit is transmitted through matchmaking push requests containing `NRSessionSearchResult` information. This means that the attacker can target anyone who joins their online session. In particular, for DS3:

- summons (`PushRequestSummonSign`)
- dark spirit invaders (`PushRequestAllowBreakInTarget`)
- players joining via covenant (`PushRequestVisit`)
- arena combattants (`PushRequestAcceptQuickMatch`)

This is already pretty bad, but the real potential is unlocked by the `RequestSendMessageToPlayers` request:

```
message RequestSendMessageToPlayers {
    repeated uint32 player_ids = 1;
    required bytes push_message = 2;
}
```

The host uses this request to directly send the `PushRequestAllowBreakInTarget` push message to invaders so that they can obtain spawn coordinates and join their P2P session. That's it. That's the only way this request is used by the game.

Yet it allows any client to send arbitrary push messages to hundreds of thousands of specific players.

I cannot stress how horribly unsafe this is. Any player can basically impersonnate the matchmaking server. By using this request to send the exploit through a `PushRequestVisit`, any online player can be remotely targeted by the attacker as long as their player ID is known. The attacker can also send the exploit to the entire online playerbase very quickly by sending multiple requests, each containing a large slice of possible player IDs.

The General Exploitation Tactic for All Games

While the RCE does not port exactly to every game, the core idea of the exploit which gives the attacker arbitrary code *redirection* is the same. If this can be achieved, it is very likely that a game-specific virtual call chain or ROP chain can then be found. This "first step" uses the following vulnerabilities:

Bug #1: No Bounds Check in Entry List Parser

Matchmaking push requests containing session join information store said information in a custom binary format which consists of a chain of length-deliminted data entries. Each entry has the following format:

```
struct Entry
{
    uint32_t type_or_id; // not sure, but probably a type (fixed length = 2, variabl
    uint32_t size;
    uint8_t data[size];
}
```

The game function responsible for copying the data of these entries blindly trusts the size field, which creates an out-of-bounds read. This can be abused by a malicious client by setting the size field to values like `0x7FFFFFFF`, causing the memory allocation to fail and the victim's game to crash. Later, this size is also passed to the constructor of a `DLMemoryInputSteam`, which is an instrumental part of the exploit.

Bug #2: Buffer Overrun in `NRSessionSearchResult` Parser

One of the entries in the data structure described above is a serialized `NRSessionSearchResult` object. The parser for this data first parses a list of properties. These properties can be 4 byte ints, 8 byte ints or null-terminated wide strings. This property list is followed by the host Steam persona name as a null-terminated wide string and some additional data not important for the exploit. Both this function and the property list parser use a fixed-size stack buffer to read strings, and in both cases no bounds check is performed on the buffer. Here is the game code responsible for copying the host name (produced using the Ghidra decompiler and then cleaned up):

```
size_t idx = 0;
wchar_t wchr = 0;
do {
  // read_wchar() function at vftable index 17 of DLEndianStreamReader
  wchr = stream_reader->read_wchar();
  player_name_buff[idx] = wchr;
  idx++;
} while (wchr != 0);
```

This leads to a buffer overrun exploit, allowing the attacker to corrupt the stack.

Paving the way for the virtual call / ROP chain

To achieve arbitrary code redirection, we use this and the memory layout of a `DLMemoryInputStream` object instantiated on the stack by the function calling the parser, which is used internally by the stream reader:

```
struct DLMemoryInputStream {
    uintptr_t* vftable_ptr; // Offset 0
    size_t data_size;       // Offset 4 (32bit) / 8 (64bit)
    uint8_t* data_buffer;   // Offset 8 (32bit) / 16 (64bit)
    // Entries after the buffer are not important for the exploit
}
```

Since we control the `data_size` field (Bug #1), it can be set to the stack memory address of the `data_buffer` field. This will succeed provided the address is constant and not too large (DS3 satisfies those requirements). Since the compiler places the stack buffer at the top of the frame, the attacker can then use Bug #2 to overwrite the lower two bytes of the `DLMemoryInputStream` 's `vftable_ptr` . Hence when the next character is read by the `DLEndianStreamReader` , it will call the `DLMemoryInputStream` internally and code will be redirected. The 2 bytes give enough leeway to jump to the 22nd function in the `DLEndianStreamReader` vftable, which calls the 6th virtual method of the object pointed at by its first field. In a 64-bit process (i.e. Dark Souls III), the following instructions would be executed:

```
MOV       RCX,qword ptr [RCX + 0x8]
MOV       RAX,qword ptr [RCX]
JMP       qword ptr [RAX + 0x40]
```

Since `RCX` is a pointer to the `DLMemoryInputStream` object, the first instruction writes the `data_size` field, which has been set to a stack address pointing to the `data_buffer` field by the attacker using Bug #1, into `RCX` . The two next instructions will thus redirect execution to the memory address the attacker has written at offset `0x40` in the data buffer. Arbitrary code redirection has now been achieved! From there the attacker can set up a chain of code redirection that copies their payload into a suitable memory region and executes it by chosing code close to virtual calls with different offsets, since the buffer now acts as a virtual method table. For the Dark Souls III proof of concept I found a setup which only requires 3 gadgets to achieve RCE:

- 0x18: `140e97700`
- 0x40: `1422be020`
- 0x68: `140e40f15`

See here for more details on these 3 gadgets. If for some other game this virtual call method is not a feasible approach, the arbitrary code redirection may still be used to setup more traditional ROP exploit.

Dark Souls III Proof of Concept Code

Running the PoC code

To run the proof of concept code, you must first have a server to connect to. While the official servers have been disabled due to the exploit, you can setup a private one using ds3os. ds3os is designed to mimic the retail server behaviour as close as possible, but security patches have already been deployed to this project to fix this exploit. However you can still setup a testing environment by building the project yourself with the `SEND_MESSAGE_TO_PLAYERS_SANITY_CHECKS` and `NRSSR_SANITY_CHECKS` constants set to `false` in BuildConfig.h. This mimics the unsafe retail server behaviour. Follow the instructions provided by ds3os to start the game and connect to your server.

Once this is done and your game is connected to the servers, build the PoC code and start the `Injector.exe` executable. It will inject a DLL containing the exploit code in the Dark Souls III process. This DLL will then use the game function that sends FRPG messages to the server in order to deliver the exploit to your own client.

Attack Vector

For the proof of concept I decided to use a `PushRequestVisit` message sent using `RequestSendMessageToPlayers` . This is the most potent version of the exploit in the sense that the target's game will immediately parse the vulnerable data after recieving it in all situations (even in the main menu).

Virtual Call Redirection Chain

Offset 0x18: 140e97700

```
LEA       RAX,[DAT_144786150]
RET
```

This gadget is used in the one at `0x68` . We need to put an address lower but fairly close to `144786998` into `RAX` ; this is the closest one.

Offset 0x40: 1422be020

```
MOV       RDX,RAX
MOV       R8,qword ptr [RCX]
CALL      qword ptr [R8 + 0x68]
```

To be able to use the gadget at offset `0x68` we need the data buffer address to be stored in `RDX` and `RCX` to stay the same. This achieves precisely that.

Offset 0x68: 140e40f15

```
; Jumping here from the gadget at offset 0x40
MOV         RBX,RDX
CMP         R9,R8

 ; Never jumps, R9 != R8
JZ          LAB_140e40f7a
MOV         RAX,qword ptr [RCX]
MOV         R8,qword ptr [RSP + 0x50]
MOV         RDX,R9
MOV         qword ptr [RSP + 0x30],RSI

; Call gadget at offset 18 (140e97700). Loads 144786150 into RAX
CALL        qword ptr [RAX + 0x18]
MOV         RSI,RAX
TEST        RAX,RAX

; Never jumps, RAX is the data buffer addr.
JZ          LAB_140e40f4d
CMP         RBP,RDI
MOV         RDX,RBX
MOV         RCX,RAX
CMOVC       RDI,RBP
MOV         R8,RDI ; RDI is a stack address close to 14F3B0, so the memcpy succeeds
CALL        memcpy

LAB_140e40f4d:
TEST        RBX,RBX
; Never jumps as RBX == RDX == data buffer addr, nonzero.
JZ          LAB_140e40f62

; We have our now fully control this RWE memory region due to the memcpy at 14478615
MOV         RCX,qword ptr [DAT_144786998]
MOV         RDX,RBX
MOV         RAX,qword ptr [RCX]
CALL        qword ptr [RAX + 0x68]
```

This gadget does almost everything for us. It calls offset `0x18` to get a memcpy destination pointer, copies our packet there and then calls the virtual function at offset `0x68` on static object at `144786998`, which we now fully control because of the memcpy call. Since the amount of memory corrupted by the memcpy is large and some regions are constantly being written to by other game threads, the exploit first loads a "setup" payload that is copied into a safe location, suspends all other threads and recopies our actual payload before jumping to it. See `rce.h` for more info.

Extra Information

I recommend checking out the source code of the proof of concept code as it has many comments detailing the structure of the packet. If you do want to see what happens at each step in real time (you should, it's pretty cool!), you can inject the proof of concept DLL while running the game under a debugger with breakpoints at the following addresses of interest:

`140ca5960`

Essentially where the exploit begins. This function is responsible for parsing the size-delimited entry list data of the PushRequestVisit message. It first extracts each entry of the list into different vectors:

```
0x140ca59f8:
    player_data_cpy_ptr = (std_vector *)VectorCopy2_140ca4ef0(&player_data_cpy,playe
    FUN_140ca5010(player_data_cpy_ptr,&spawn_data,0x1c);
    FUN_140ca5010(player_data_cpy_ptr,&unk,4);
    FUN_140ca4fa0(player_data_cpy_ptr,&nrssr_data);
```

The function `140ca5010` check entry sizes, but `140ca4fa0` is for variable size entries and does not perform sanity checks on the size field (Bug #1). To achieve the arbitrary code redirection exploit described above we need to set it to `14F3B0`. This will cause an out-of-bounds read of approx. 1.3MiB but the memory page should be large enough to avoid access violations.

`140ca56b0`

This function is called by the previous one with `nrssr_data` as an argument. Creates the `DLMemoryInputStream` object on the stack which is then passed as an argument to the NRSSR parser.

`141955f50: ParseNRSessionSeachResult`

The `NRSessionSearchResult` parser. Verifies NRSSR signature and version numbers (`14196a0f0`), parses the property list (`14196a260`), host name (`14195603a`) and some more info (see rce.h)

`14195603a`

Loop in the above function that unsafely copies the host name (Bug #2). Here are some addresses that may help keeping track of what is happening during the buffer overflow:

- Parser stack buffer address: `14F128`
- `DLMemoryInputStream` stack address: `14F3A0`
- `DLMemoryInputStream` vtable pointer after the overwrite: `1439e8b30`
- Offset of the virtual function in the `DLMemoryInputStream` used by the `DLInputStreamReader` : `0x18`

1439e8b48

```
MOV       RCX,qword ptr [RCX + 0x8]
MOV       RAX,qword ptr [RCX]
JMP       qword ptr [RAX + 0x40]
```

Where we end up after the first code redirection caused by the overwritten memory stream vftable. This is where the chain of virtual call redirections begin.

1. For Dark Souls III Ver. 1.15. The maximum theoretical payload size depends on the stack layout and as such will vary by game and version. ↩

Releases

No releases published

Packages

No packages published

Languages

● C++ 100.0%