EXPLORING COUNTER-STRIKE: GLOBAL OFFENSIVE ATTACK SURFACE

Rédigé par Victor Cutillas , Louis Jacotot - 08/01/2024 - dans Exploit , Reverse-engineering

Back in 2021, we studied the attack surface of Counter-Strike: Global Offensive as a side research project. We found and reported a relative heap out-of-bounds write vulnerability triggerable remotely, impacting code that is no longer present with the release of Counter-Strike 2. In fact, no patch was released in the meantime despite multiple follow-ups. We share today the details of this bug and our research about the attack surface and generic exploitation primitives.

Back in 2021, during our 6-month reverse-engineering internship, we studied the attack surface of *Counter-Strike: Global Offensive* (CS:GO) for approximately 25 days, as a side research project that was different from our main internship subject. Like other games such as *Half-Life: 2* and *Team Fortress 2*, CS:GO is based on Source Engine, a multiplayer-able 3D game engine created by Valve, first released in 2004. For our research, we set up a lab with a Windows client and a Linux server version *1305 1.37.9.6* (July 21, 2021) and retrieved the 2017 version of the source code available through public GitHub repositories.

ATTACK SURFACE OVERVIEW

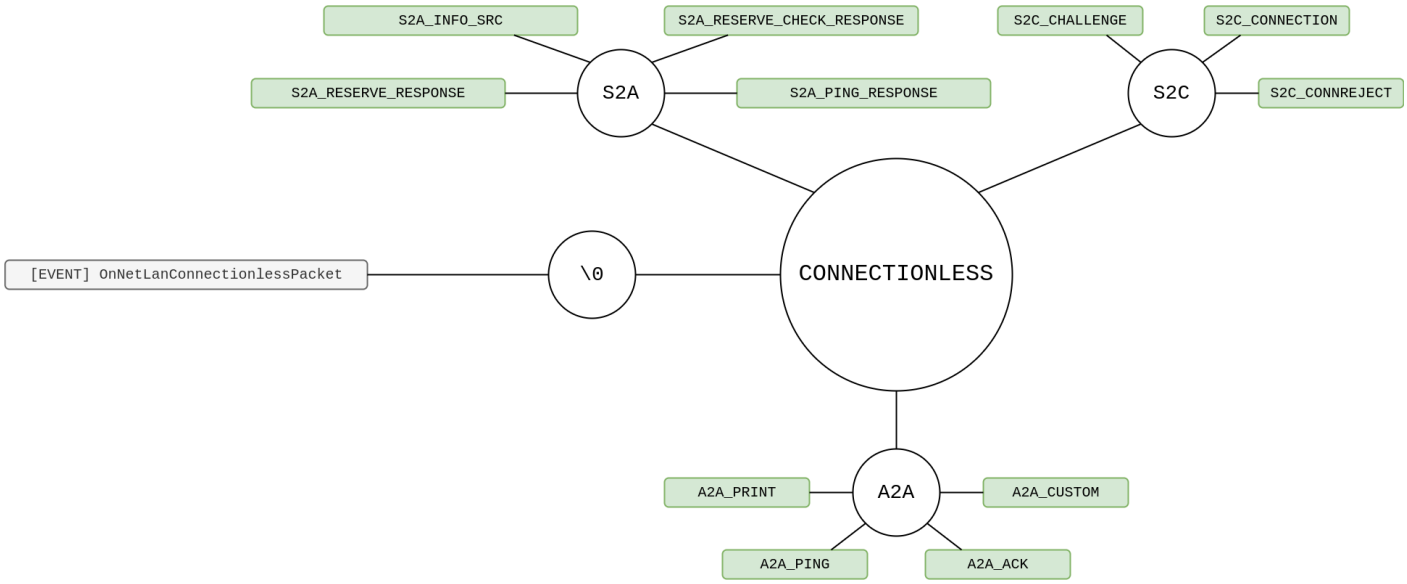
The remote attack surface of CS:GO is quite complex as it supports a lot of features. Interesting components to attack include not only game-specific network and message logic, remote file downloads and parsers for additional content, but also voice codecs which are broadcasted client-to-clients by the server. Moreover, an in-game *Chromium Embedded Framework* Web browser is used to display a message of the day (MOTD) popup from a server-controlled URL. Source Engine also supports a Remote Console (RCON) for server administration, Half-Life TV (HLTV) for in-game spectating and the Panorama UI framework, leaving the door open to nice XSS[1].

With all that attack surface, and given that the research has to start somewhere, we chose to study game networking first. Source Engine network transmission is done over UDP: connectionless packets – used to retrieve information about a server or establish a connection – are sent as raw datagram packets, without any additional layer. However, in-game data is sent over a proprietary `NetChannel` layer supporting encryption, fragmentation, and compression of reliable, ordered packets. In this game engine architecture, the server is authoritative, thus we decided to focus on client-side vulnerability research as the surface is way larger compared to the server-side.

CONNECTIONLESS MESSAGES

Due to its simplicity, the connectionless attack surface is quite interesting to study: a simple loop in `NET_ProcessSocket()` polls and forwards incoming datagrams starting with a `\xff\xff\xff\xff` header to `ProcessConnectionlessPacket()`. Then, a command number is read and processed by a specific handler, according to the following implementations where `A` =any, `S` =server, `C` =client:

1. Server implements `A2A` , `A2S` and `C2S` command groups.
2. Client implements `A2A` , `S2A` , `S2C` command groups.
3. The special command `\0` is used in the client implementation to transmit matchmaking messages in LAN networks as a `OnNetLanConnectionlessPacket` event.



Client connectionless surface

In particular, the Steam server browser implementing some Source Engine client commands was vulnerable to a stack-buffer overflow in the `S2A_PLAYER` handler, in 2018[2].

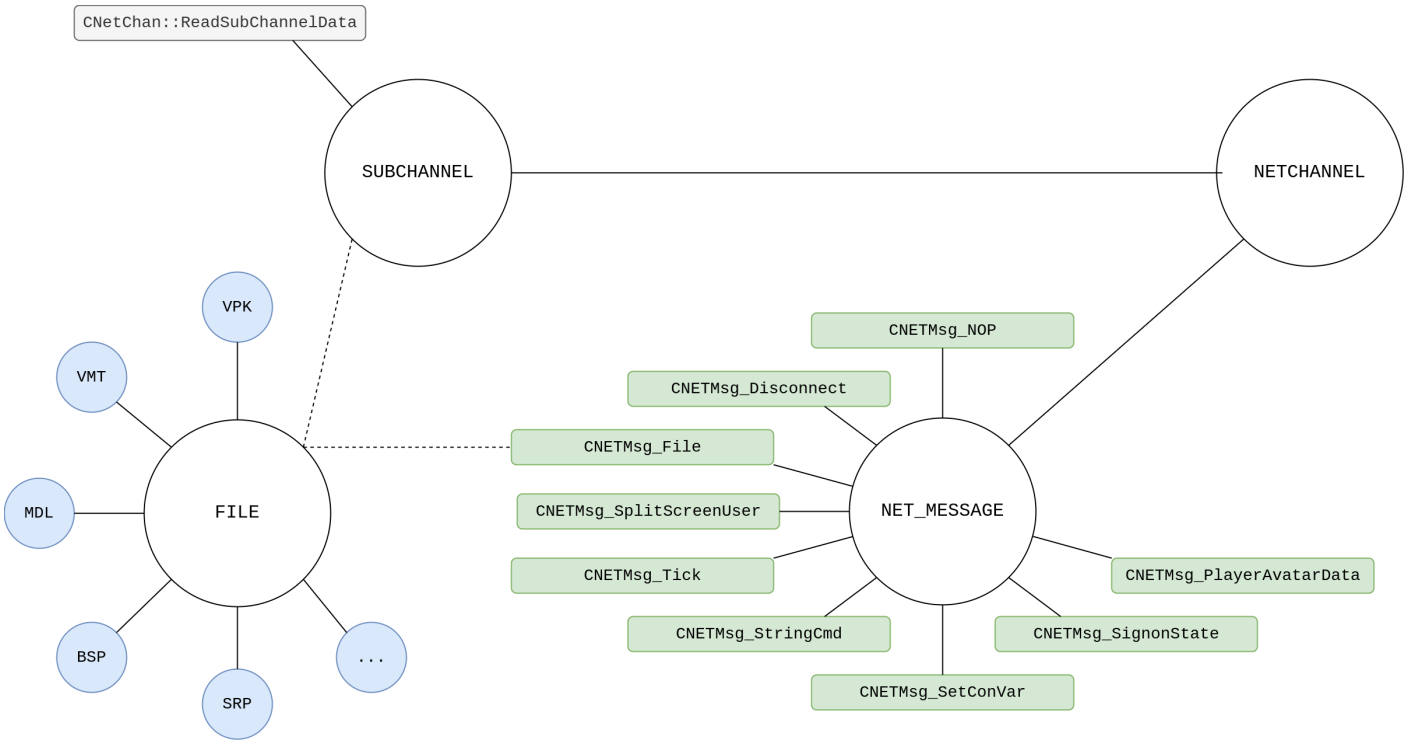
NETWORK CHANNELS

Netchannel traffic is encrypted with the ICE block cipher[3]. It turns out a fixed key is used on CS:GO community servers, derived from the build version number and using a simple algorithm:

```
N = int(BUILD.replace(b'.', b'')) # BUILD = b'1.37.9.6'
key = b'CSGO' + bytes([(N >> k) & 0xff for k in [0, 8, 16, 24, 2, 10, 18, 26, 4, 12, 20, 28]])
```

Upon decryption, the packets are handled by `CNetChan::ProcessPacket()` and the custom transport layer logic is applied: duplicated or out-of-order packets are discarded, ACKs are sent... Next, packets flagged as reliable are processed, in a *subchannel*, supporting fragmentation and compression in `CNetChan::ReadSubChannelData()`. Then, in `CNetChan::ProcessMessages()`, the fun part of the networking stack begins: a message type is decoded using a 32-bit *ZigZag* algorithm[4] and the payload is deserialized – with *protobuf* on CS:GO and as a bitstream on other Source Engine branches.

The implementation of `NET_MESSAGE` messages is shared between client and server. In addition, files sent – and then parsed – via subchannels account for a significant part of the attack surface. For example, corrupted BSP maps[5] and unrestricted VPK archive uploads[6] have already been shown to lead to RCE.



Client netchannel surface

SERVER NETCHANNEL

The `SVC_MESSAGE` server netchannel message category includes ~30 server-to-client messages and encapsulates other game features, which results in a large attack surface:

- **User messages**: the `CSVCMsg_UserMessage` subcategory encapsulates another layer of protobuf-serialized payload and is used by CS:GO to implement game-specific logic, such as votes.
- **Game events**: the `CSVCMsg_GameEvent` message is used to dispatch `KeyValues` events.
- **Entities netcode**: entity-specific events are transmitted using a `CSVCMsg_EntityMsg` message. In addition, the `CSVCMsg_PacketEntities` subcategory encapsulates a bitstream-serialized payload to handle entities *netcode*.

VULNERABILITY RESEARCH

FUZZING

While reading the source code, fuzzers were run using AFL++[7] against file formats that can be loaded and transmitted from server to clients: MDL and BSP. Both are proprietary and used to respectively define 3D models and maps for Source Engine-based games.

The harness used was extracted from a blogpost of *niklasb*[8] about a challenge from *RealWorldCTF 2018*, and slightly modified to fit our needs. It targets the Linux server implementation.

RELATIVE HEAP OUT-OF-BOUNDS WRITE

After two nights of fuzzing, around a hundred unique crashes were triggered. About 95% of them were due to out-of-bounds read accesses from unchecked offsets and did not look interesting. However, a few incorrect write crashes in `datacache.so` also occurred when parsing MDL files: they were all side effects of the same root cause bug. The vulnerable code is present in the source code in `datacache/mdlcache.cpp`, at the entrypoint of MDL file parsing.

```
//-----------------------------------------------------------------------------
// Attempts to load a MDL file, validates that it's ok.
//-----------------------------------------------------------------------------
bool CMDLCache::ReadMDLFile( MDLHandle_t handle, const char *pMDLFileName, CMDLCacheData &cacheData ){
  bool bOk = cacheData.ReadFileNative( pFileName, "GAME" );
  // [...]
  studiohdr_t *pStudioHdr = (studiohdr_t*)cacheData.Data();
  // [...]
  if ( pStudioHdr->studiohdr2index == 0 )
  {
    DevWarning( "Model %s doesn't have a studiohdr2, which should've been fixed. This is required of all m
odels now.\n", pMDLFileName );
    return false;
  }
  // critical! store a back link to our data
  // this is fetched when re-establishing dependent cached data (vtx/vvd)
  pStudioHdr->SetVirtualModel( MDLHandleToVirtual( handle ) );
  // [...]
}
```

The `studiohdr_t` structure definition is available in file `public/studio.h`:

```
struct studiohdr_t
{
  int id;
  int version;
  // [...]
  int studiohdr2index;

  studiohdr2_t* pStudioHdr2() const
  {
    return (studiohdr2_t *)( ( (byte *)this ) + studiohdr2index );
  }
  // [...]
  void SetVirtualModel( void* ptr )
  {
    Assert( studiohdr2index );
    if ( studiohdr2index ) { pStudioHdr2()->virtualModel = ptr; }
  }
}
```

And the conversion function is defined in `public/datacache/imdlcache.h`:

```
typedef unsigned short MDLHandle_t;
// [...]
inline void* MDLHandleToVirtual( MDLHandle_t hndl )
{
  return (void*)(uintp)hndl;
}
```

`pStudioHdr->studiohdr2index` is directly read from the MDL file and used to write a value without any bound checking: this is a **relative heap out-of-bounds write** vulnerability. The `MDLHandle_t` handle method parameter is zero-extended to `void *` before being written in memory. As a result of the unsigned short to pointer expansion, the two most significant bytes are null. Besides that, the value of the two least significant bytes is dependent on the context of the `CMDLCache` as it is linked to the number of previously cached MDL objects, which is unknown to the attacker.

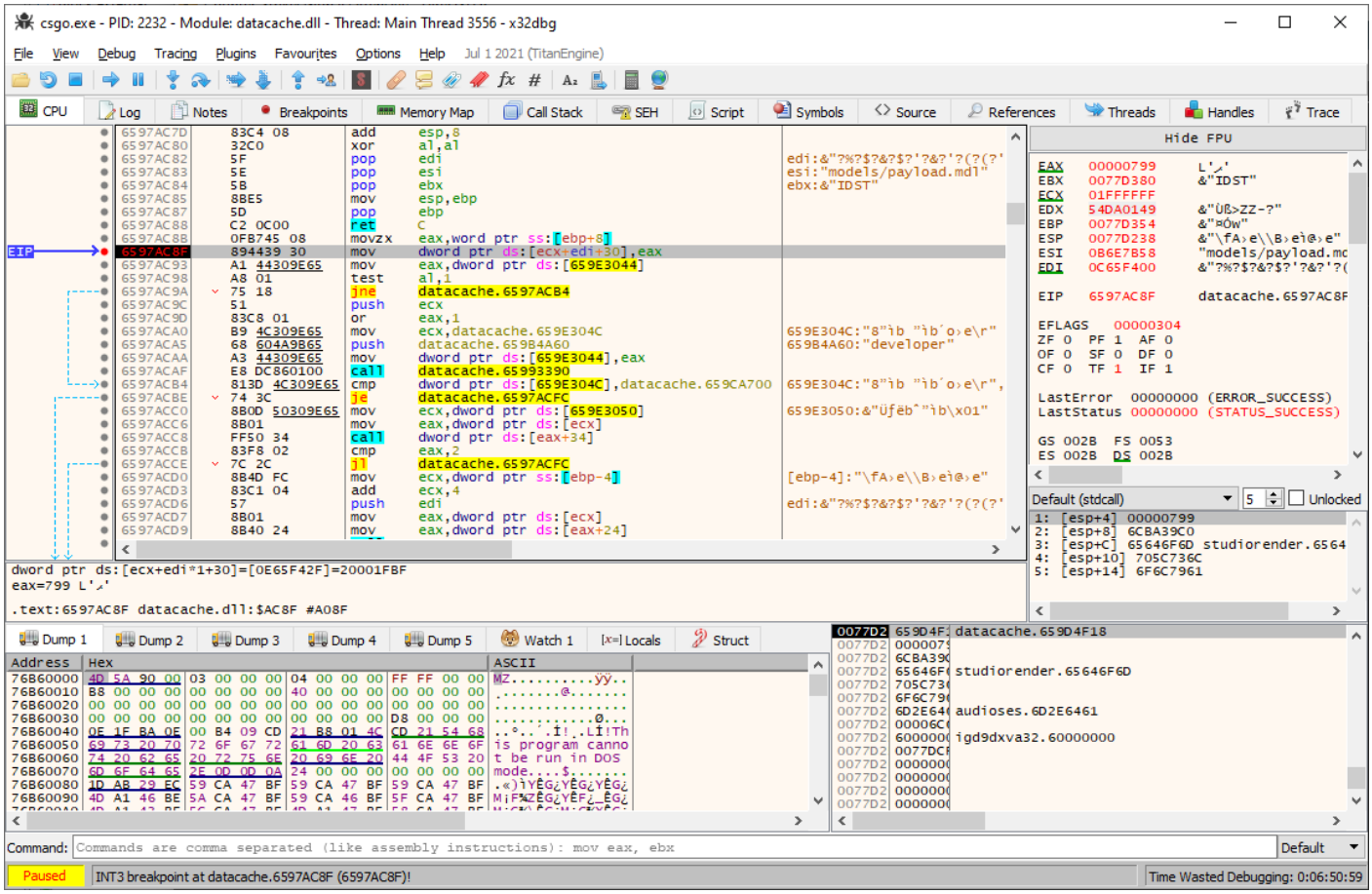
Building a proof of concept for this bug is very straightforward: setting `studiohdr2index` to some huge offset will most likely trigger access to unmapped memory. Here is a hexadecimal dump of a minimal payload file with the index set to `0x1ffff00d`, which usually leads to a crash during the call to `studiohdr_t::SetVirtualModel()`:

```
00000000  49 44 53 54 00 00 00 00  00 00 00 00 00 00 00 00  |IDST............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000190  0d f0 ff 1f 00 00 00 00                           |........|
00000198
```

Reproducing the bug is possible using the following steps:

1. Copy payload file to `C:\Program Files (x86)\Steam\steamapps\common\Counter-Strike Global Offensive\csgo\models\payload.mdl`.
2. Open the console during an offline game, enable flag `sv_cheats 1`.
3. Load the model with command `prop_dynamic_create payload`.

The root cause of the crash can be confirmed by setting a breakpoint on the inlined `studiohdr_t::SetVirtualModel()` call: `ECX` register contains the offset stored in the file, and `EAX` represents the identifier of the virtual model being loaded.



As the primitive was pretty limited and research just started, we had hope to find some other memory corruption primitive and started manual code source review.

REVERSE-ENGINEERING AND SOURCE CODE AUDIT

In 2017, the source code was uploaded online and is still available through public GitHub repositories, but has obviously been updated since. We used it for the audit and proceeded to compare some of it with the compiled up-to-date version. The codebase is old-school C++ and C code and some dangerous patterns emerge when looking at the code.

First, there are a lot of uses of the signed `int` type, pretty much everywhere where a size is being represented: this carries incorrect semantic information – why would you use a signed type to represent a size? – and may lead to bugs where the developer does not think about a size variable being possibly negative, such as when checking an out-of-bound index.

Very few bound-checking is done in the code, which have been the cause of many bugs such as a RCE when handling `CSVCMsg_SplitScreen` messages[9]. A lot of C-style functions are being called to manipulate strings, copy memory, etc. We rediscovered 13 out-of-bounds bugs by reading through the source code, and figured out that some fixes are actually quite shallow: bound checks are sometimes done inside the callers of functions responsible for fetching data and not in the getters themselves. This is a pretty dangerous pattern as forgetting to patch a single call path means that the vulnerability is still likely to be reachable, however this has not been observed in actual fixes.

In addition, during source code audit, we identified two stack-buffer overflows with controlled size and data, in the pre-authentication Connectionless Messages feature. Indeed, the `S2C_CHALLENGE` command handler first reads two sizes from the received message `[1][2]`, and then copies the corresponding bytes size to local variables, stored on the stack `[3][4]`.

```
bool CBaseClientState::ProcessConnectionlessPacket(netpacket_t *packet)
{
        /* [...] */
        bf_read &msg = packet->message;
        /* [...] */
        int c = msg.ReadByte();
        switch (c) { /* [...] */
        case S2C_CHALLENGE: /* [...] */
                byte chKeyPub[1024] = {};
                byte chKeySgn[1024] = {};

                int cbKeyPub = msg.ReadLong(); /* [[1]] */
                msg.ReadBytes(chKeyPub, cbKeyPub); /* [[3]] */

                int cbKeySgn = msg.ReadLong(); /* [[2]] */
                msg.ReadBytes(chKeySgn, cbKeySgn); /* [[4]] */
                /* [...] */
        }
}
```

However, after reverse-engineering the corresponding up-to-date binary code, we noticed that these simple vulnerabilities were already fixed.

In the end reviewing the source code did not result in any interesting vulnerability, yet it allowed to discover some exploitation primitives described further below.

EXPLOITING THE VULNERABILITY

REMOTELY TRIGGERING THE MODEL PARSING BUG

In order to remotely trigger the bug on a client connecting to a malicious server, the latter must force the client to load a custom MDL file. When the client is loading the current map while connecting to the server, the server transmits a string table called `downloadables`, describing the resources required to play. The client will download missing resources from the server using `CNETMsg_File` requests only if the configuration directive `sv_allowupload` option is enabled. It is important to note that since February 2018, the default parameters for the CS:GO client is to disable this option. The good news is that community servers usually require clients to enable it in order to load custom maps.

One way of remotely loading the model would be to create an entity referring to the malicious model, and arbitrarily render it to the client at any moment of the game in order to trigger the load. However, creating an entity is tedious, and we found another easier way of triggering the vulnerability by reading the client source code handling the `downloadables` string table.

Client consistency checks on downloaded files can be enforced by the server through network-transmitted boolean configuration option `sv_consistency`. Those checks are implemented in `CClientState::ConsistencyCheck()`, called when the sign-on state reaches value `SIGNONSTATE_NEW`. This method iterates over the transmitted filenames to perform either CRC or model size checks, depending on server-transmitted metadata. Files tagged `CONSISTENCY_EXACT` will have their CRC

checked, whereas those tagged `CONSISTENCY_BOUNDS` will be loaded as models in order to perform size checks. A consistency check error will disconnect the client from the server.

We developed a proof of concept by instrumenting the Linux server using Frida: the method `CDownloadListGenerator::SetStringTable()` is hooked to alter the `downloadables` string table by adding a malicious model tagged `CONSISTENCY_BOUNDS` to trigger the vulnerability remotely. Upon connection, the client loads server resources and successfully crashes before the loading screen ends.

EXPLOITATION SCENARIO

Months after the initial vulnerability research, we have decided to look back into the only bug found looking viable for exploitation. One week was dedicated to this analysis: the aim was not to get a working exploit but rather deduce how much time would be needed in order to achieve remote code execution on CS:GO Windows client.

First, basic security mitigations are applied to every PE and DLL file shipped with the Windows client:

- ASLR enabled (`IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE`)
- DEP enabled (`IMAGE_DLLCHARACTERISTICS_NX_COMPAT`)
- No stack smashing protection

The only available primitive being a relative heap out-of-bound write, we need several steps to enhance our control over client's memory. Here is an example of typical steps required to exploit this kind of bug:

1. Find a reliable heap massaging technique to get an interesting object next to our `studiohdr_t` .
2. Gain a relative memory leak primitive by partially overwriting some size / pointer in order to bypass ASLR.
3. Get an arbitrary memory read primitive – preferably without massaging once again to maximize reliability.
4. Abuse the bug once again to get a write-what-where primitive.
5. Gain remote code execution.

Crafting a working exploit is a whole other mess. The very same heap is massively used by complex software components using default *Windows 10* memory allocator: for instance, CS:GO embeds the V8 JavaScript engine along with a shader compiler and Valve GUI framework which all heavily manipulate the main heap state. Getting a state deterministic enough to reliably get interesting objects next to our `studiohdr_t` does not look like an easy job.

We ended up with the following attack scenario:

1. Client with `sv_allowupload 1` connects to an attacker-controlled server.
2. During resource loading, server enforces bounds-check for a custom MDL.
3. Client downloads custom MDL from the server.
4. Client loads it to check against previously given server bounds.
5. The bug is triggered, corrupting client's memory and handing back full control of the client computer to the server.

AVAILABLE PRIMITIVES

A lot of time had already been allocated for the research and no other interesting bugs were found. We were therefore constrained to deal with this poor man's bug, and decided to focus on discovering exploitation primitives even though time would be lacking to actually implement an exploit.

HEAP SPRAYING

It is possible for the server to authoritatively update a player avatar by sending a message of type `CNETMsg_PlayerAvatarData` , defined as follows in `common/netmessages.proto` :

```
message CNETMsg_PlayerAvatarData
{ // 12 KB player avatar 64x64 rgb only no alpha
```

```
    // WARNING-WARNING-WARNING
    // This message is extremely large for our net channels
    // and must be pumped through special fragmented waiting list
    // via chunk-based ack mechanism!
    // See: INetChannel::EnqueueVeryLargeAsyncTransfer
    // WARNING-WARNING-WARNING
    optional uint32 accountid = 1;
    optional bytes rgb = 2;
}
```

An arbitrary number of bytes can be sent using this message, the `rgb` field being typed as `bytes`. The following code corresponds to the message handler on the client:

```
// engine/baseclientstate.cpp
bool CBaseClientState::NETMsg_PlayerAvatarData( const CNETMsg_PlayerAvatarData& msg )
{
    PlayerAvatarDataMap_t::IndexType_t idxData = m_mapPlayerAvatarData.Find( msg.accountid() );
    if ( idxData != m_mapPlayerAvatarData.InvalidIndex() )
    {
        delete m_mapPlayerAvatarData.Element( idxData );
        m_mapPlayerAvatarData.RemoveAt( idxData );
    }

    CNETMsg_PlayerAvatarData_t *pClientDataCopy = new CNETMsg_PlayerAvatarData_t;
    pClientDataCopy->CopyFrom( msg );
    m_mapPlayerAvatarData.Insert( pClientDataCopy->accountid(), pClientDataCopy );

    return true;
}
```

Server handling is pretty straightforward, first deleting the player avatar if it already exists, then allocating a new one and inserting it in a map. Any arbitrary account ID is accepted, allocations are persistent and size is attacker-controlled, which makes it a powerful heap spraying primitive.

WRITE TO A FIXED MEMORY LOCATION

Another useful primitive for a reliable exploit is having the ability to write arbitrary data to a fixed memory location, such as a variable located in a `.data` section in the loaded executable or a library. Such a primitive is easily usable by the server, through the `CCSUsrMsg_ShowMenu` message. Function `CHudMenu::MsgFunc_ShowMenu()` handles the incoming message on the client side:

```
// game/client/menu.cpp
bool CHudMenu::MsgFunc_ShowMenu( const CCSUsrMsg_ShowMenu &msg)
{
    // [...]
    if ( m_bitsValidSlots )
    {
        Q_strncpy( g_szPrelocalisedMenuString, msg.menu_string().c_str(), sizeof( g_szPrelocalisedMenuStri
ng ) );

        GetClientMode()->GetViewportAnimationController()->StartAnimationSequence("MenuOpen");
        m_nSelectedItem = -1;

        // we have the whole string, so we can localise it now
        char szMenuString[MAX_MENU_STRING];
        Q_strncpy( szMenuString, ConvertCRtoNL( hudtextmessage->BufferedLocaliseTextString( g_szPrelocalis
edMenuString ) ), sizeof( szMenuString ) );
        g_pVGuiLocalize->ConvertANSIToUnicode( szMenuString, g_szMenuString, sizeof( g_szMenuString ) );
        // [...]
    }
    else
```

```
    {
        HideMenu();
    }

    return true;
}
```

As shown in this code, a global variable called `g_szPrelocalisedMenuString`, defined as a `char[512]` string in `game/client/menu.cpp`. Despite this variable being treated as a null-terminated string in the code, writing it multiple times still allows the server to craft arbitrary data in a section of library `engine.dll`.

CONCLUSION

Overall code is legacy and does not implement in-depth security protections. A lot of crashes have been quickly triggered by fuzzing BSP and MDL format parsers but most of them were not relevant for exploitation. The bug found does not provide a very strong primitive, but a motivated attacker is believed to be able to gain remote code execution. It relies on the configuration directive `sv_allowupload 1` being enabled, which seems pretty widespread among CS:GO players to play on community servers.

Reporting the bug to Valve through *HackerOne* managed program was a long process, as shown in the timeline available below. The ticket was closed with the release of *Counter-Strike 2* and the impacted code is no longer present. In fact, to our knowledge, no patch was released in the meantime, despite multiple follow-ups.

TIMELINE

- **2022.07.11**: Initial report sent on HackerOne Valve's bug bounty program.
- **2022.07.15**: [H1] triage acknowledges and requests detailed reproduction steps.
- **2022.07.21**: Reply with detailed reproduction steps.
- **2022.10.03**: Ask for an answer.
- **2022.10.04**: [H1] triage cannot reproduce, asks us to deploy a PoC server.
- **2022.10.19**: Reply by saying that deploying a server is too much work.
- **2022.10.25**: [H1] triage asks to update to latest CS:GO release.
- **2022.11.18**: Reconfirm bug on latest update & provide a simpler way to locally test the bug.
- **2022.11.30**: [H1] triage validates and submits to Valve remediation team.
- **2023.01.05**: Ask for an update.
- **2023.01.30**: [H1] triage says no update have been received.
- **2023.02.13**: Ask for an update.
- **2023.02.27**: Ask for an update.
- **2023.03.02**: [H1] says Valve is working on this.
- **2023.04.25**: Ask for an update.
- **2023.04.25**: [H1] rewards $750.
- **2023.07.10**: Ask for an update for the fix.
- **2023.09.19**: Ask for an update for the fix.
- **2023.10.16**: Inform about the potential full disclosure of the bug.
- **2023.10.16**: [H1] warns about violation of terms and asks to wait.
- **2023.10.26**: [H1] closes the issue and marks it as resolved.
- **2024.01.08**: Publication of this blog article.

---

1. https://hackerone.com/reports/631956
2. https://hackerone.com/reports/470520
3. https://en.wikipedia.org/wiki/ICE_(cipher)

4. https://protobuf.dev/programming-guides/encoding/#signed-ints
5. https://hackerone.com/reports/351014
6. https://nyancat0131.moe/post/source-engine/vpk-upload/
7. https://aflplus.plus/
8. https://phoenhex.re/2018-08-26/csgo-fuzzing-bsp
9. https://secret.club/2021/05/13/source-engine-rce-join.html