StarCraft: Remastered
Emulating a buffer overflow for fun and profit
A note before we begin

Blizzard Entertainment in no way endorses or condones reverse engineering of our properties.

The exercises herein were conducted to understand the methods used to create unlicensed behaviors.
About Me: Elias Bachaalany

• Anti-Cheat Engineer, Blizzard Entertainment

• Previously worked at Hex-Rays and Microsoft

• Technical writer:
  • Practical Reverse Engineering, Antivirus Hackers Handbook
  • Batchography

• Passionate about reverse engineering and low-level programming on MS-Windows

• Interested in debuggers, emulators, API hooking, dynamic binary instrumentation and virtualization technologies

• Contact
  • Email: ebachaalany at blizzard.com
  • Twitter: @0xeb
Comrades on the adventure

My colleagues
• Guillaume Breuil, Yi Deng, Chris Genova, Mark Chandler, James Touton, Pete Stilwell, Zak Bennett and Grant Davies

Tools
• SCMDraft2 map editor - Henrik Arlinghaus
• trgk (Trigger King) - https://github.com/phu54321/
• MPQ tools – Ladislav Zezula
• BWAPI - Adam Heinermann
• IDA Pro - Hex-Rays
• Diaphora – Joxean Koret
• EUDEnabler and the EUDDB - Farty1Billion - http://farty1billion.dyndns.org/EUDDB/

South Korean map makers and tools community
• Kongze1004 – Random Tower Defense map author
• Sksljh2091 – Mario Exodus map author
• Jacksell12, Deation, Sato

Community Sites
• TeamLiquid, StarEdit Network, Naver.com

Sorry if I missed anyone!
Backstory /1

• StarCraft is a science fiction RTS (real-time strategy)

• Released for PC and Mac on March 31, 1998

• StarCraft: Brood War - Expansion pack released on November 30, 1998

• Significant patches to this talk:
  • 1.16.1 - 01/21/2009 – Last patch for 8 years
  • 1.18.0 - 04/18/2017 – First modern patch
  • 1.20.0 – 08/14/2017 – StarCraft: Remastered
  • 1.21.0 – 12/07/2017 – EUD reintroduced via emulation
StarCraft had various buffer overflow bugs, but one was related to a particular trigger condition and action:
- The Extended Unit Death trigger
  - Or simply: EUD

Blizzard did not update StarCraft between 2009 and early 2017
- The community re-enabled the bug with custom launchers and tools

Patch 1.17 was slated for release but was held back because it would break mods, tools, and launchers:
- wMode
- wLauncher, ChaosLauncher
- BWAPI – Plugin to write AI bots that play StarCraft
StarCraft maps based on EUD triggers thrived among the South Korean map makers community.

The EUD triggers:
- Are encoded in the map file
- Allowed arbitrary memory read and write:
  - The majority of the public EUD maps in circulation have hardcoded addresses compatible with StarCraft 1.16.1 on Windows
  - I am not aware of any EUD maps for the MacOS version of the game

The EUD exploit allowed modders to author maps that modify the game radically:
- Random Tower Defense
- Mario Exodus Map
- Etc.

Backstory /3
Random Tower Defense – EUD map
Bouncing Ball EUD map (SC 1.16.1)
Bouncing Ball EUD map (SC:R w/ emulation)
스타크래프트 유즈맵 [EUD] Mario EXODUS V0.4 (StarCraft Use map)
• The Mario Exodus map author created a level editor!
• The map was developed using trgk’s epScript language and compiler
StarCraft map file format

- They are just MPQ archives
  - The MPQ format has been extensively reverse engineered and documented by the community

- They contain various files:
  - They contain custom WAV audio used by the map
  - `staredit/scenario.chk` ← The actual map chunk file
    - This file contains the triggers chunk
    - It contains strings table chunk
    - It contains a chunk describing buildings and units
  - Etc.
Map file in MPQ Editor

- Ladik’s MPQ editor can be used to view or modify the contents of an MPQ map file

Note the chunk file: “staredit/scenario.chk”
Scenario chunk file /1

- Made of one or more chunks:

```c
typedef struct {  
    union {  
        BYTE FourCC[4];  
        DWORD ckID;  
    };  
    DWORD ckSize;  
} CK_HDR;
```

- Chunk header is followed by the chunk body
- The game parses each chunk based on its ID:

```c
static const CHUNKTABLE Bwar100Pass2_MapSettings[] = {
    {{'S', 'T', 'R', 'S'}}, maphdr_STR, true},
    {{'M', 'T', 'X', 'M'}}, maphdr_MTXM, true},
    {{'T', 'H', 'G', '2'}}, maphdr_THG2, true},
    {{'M', 'A', 'S', 'K'}}, maphdr_MASK, true},
    {{'U', 'N', 'I', 'S'}}, maphdr_UNIS_Bwar, true}, // Brood War handler
    {{'U', 'P', 'G', 'S'}}, maphdr_UPGS_Bwar, true}, // Brood War handler
    {{'T', 'E', 'C', 'S'}}, maphdr_TECS_Bwar, true}, // Brood War handler
    {{'P', 'U', 'N', 'I'}}, maphdr_PUNI, true},
    {{'P', 'U', 'P', 'G'}}, maphdr_PUPG_Bwar, true}, // Brood War handler
    {{'P', 'T', 'E', 'C'}}, maphdr_PTEC_Bwar, true}, // Brood War handler
    {{'U', 'N', 'I', 'T'}}, maphdr_UNIT, true},
    {{'U', 'P', 'R', 'P'}}, maphdr_UPRP, true},
    {{'M', 'R', 'G', 'N'}}, maphdr_MRGN_Ext, true},
    {{'T', 'R', 'I', 'G'}}, maphdr_TRIG, true},
    {{'C', 'O', 'L', 'R'}}, maphdr_COLR, true}, // new chunk
};
```
Scenario chunk file /2

- Some chunks might have their own sub-headers
- The strings chunk is such an example:

```c
typedef struct TStrTbl {
    UWORD wStrCount; // number of strings in table
    UWORD wStrOffsets[1]; // variable number of str pointers
    // variable length string data follows pointers
} TStrTbl, * TPStrTblPriv;

// STR - MAP STRING TABLE
#pragma pack(push, 1)
static BOOL CALLBACK maphdr_STR(HCHUNK hChunk, DWORD dwSize, LPARAM data) {
    gpMapStrs = (TPStrTbl)ALLOC(dwSize);
    if (!gpMapStrs)
        return false;
    gdwMapStrSize = dwSize;
    if (!ReadChunk(hChunk, gpMapStrs))
        return false;
    return true;
}
#pragma pack(pop)
```
Scenario chunk file /3

- The strings chunk can be used to hide data not used by the game directly
  - When CK_HDR.ckSize > (sizeof(the complete TStrTbl header) + \( \sum \) strlen(of all strings in the table) )
- The modders hide additional triggers in the cave area of the string chunk
Scenario chunk file /4

- This screenshot shows the last string in the strings table
- That’s not the chunk’s end though, it is just the string table’s end
- The remaining bytes are additional triggers inserted by the EUD trigger compiler

https://github.com/phu54321/
What are triggers? /1

• They are a set of conditions and actions that get evaluated during the game loop

• There are trigger conditions that tell you when:
  • A certain time period has elapsed (timers)
  • Player resources reached a certain amount
  • A map location has been reached
  • Etc.

• When all the trigger conditions are fulfilled, then you can do actions such as:
  • Play WAV file
  • Display a message
  • Create, kill, move a unit, etc.
  • Change unit owner and health points
  • Give player resources
  • Etc.
What are triggers? /2

- Triggers are stored inside the map chunk file.
- The triggers chunk is simply an array of _trigger structs.
- Each trigger has an array of the CONDITION and ACTION structures.

- The dwPlayer and wType fields are user controlled:
  - They are used to read/write out-of-bounds inside an array.
- The bOpCode field dictates the trigger condition and action type.
What are triggers? /3

- The **bOpCode** field is used to select which condition or action to execute:

```c
ConditionFcn sgConditionFcns[NUM_CONDITION_CODES] = {
    cond_always,
    cond_timer,
    cond_control,
    ...
    cond_opponents,
    cond_deaths,
    cond_least_control,
    cond_least_control_atloc,
    cond_least_kills,
    cond_lowest_score,
    cond_least_resource,
    cond_score,
    cond_always,
    cond_never,
};
```

```c
ActionFcn sgActionFcns[NUM_ACTION_CODES] = {
    act_none,
    act_victory,
    act_defeat,
    ...
    act_doodad,
    act_invincible,
    act_create_unit,
    act_set_deaths,
    ...
    act_set_unit_res,
    act_set_hangar,
    act_stop_timer,
    act_start_timer,
    act_draw,
    act_alliance,
    act_disable_escape,
    act_enable_escape,
};
```
What are triggers? /4

- Each trigger condition is evaluated, then the actions are performed if all conditions succeed:

```c
static void trigger_parse(LISTPTR(TRIGGERNODE) pTrigList) {
    // parse the conditions for each trigger
    ITERATELISTPTR(TRIGGERNODE, pTrigList, pTrigger) {
        // has this trigger already executed?
        if (pTrigger->t.lFlags & TF_COMPLETE) continue;

        sgpCurrTrigger = pTrigger;
        if (! (pTrigger->t.lFlags & TF_EXECUTING)) {
            // if the trigger isn't already executing, parse the conditions
            if (! trigger_cond_parse(pTrigger))
                continue; // conditions were not met
            pTrigger->t.bCurrAction = 0; // start with action 0
        }

        // execute the trigger's actions
        trigger_execute(pTrigger);
    }
}
```
What are triggers? /5

```c
static int trigger_cond_parse(PTRIGGERNODE pTrigger) {
  PCONDITION pCond;

  for (int i = 0; i < MAX_CONDITIONS; i++) {
    pCond = &pTrigger->t.tConditions[i];
    // the map editor can be used to disable a condition
    if (pCond->bFlags & CF_DISABLED)
      continue;
    // no condition indicates the end of the list
    if (pCond->bOpCode == COND_NONE)
      break;
    // call the function associated with the current condition
    app_assert(pCond->bOpCode < NUM_CONDITION_CODES);
    if (sgConditionFcns[pCond->bOpCode](pCond))
      continue;
    // one condition failed -- no need to check the rest
    return FALSE;
  }
  // all conditions were met
  return TRUE;
} « end trigger_cond_parse »
```

```c
static void trigger_execute(PTRIGGERNODE pTrigger) {

  // execute trigger actions until an action doesn’t complete
  int result = 1;
  while ((result && (pTrigger->t.bCurrAction < MAX_ACTIONS))) {
    pAct = &pTrigger->t.tActions[pTrigger->t.bCurrAction];
    ...
    ...
    // no action indicates the end of the list
    if (pAct->bOpCode == ACT_NONE) {
      pTrigger->t.bCurrAction = MAX_ACTIONS;
      break;
    }
    // call the action function
    app_assert(pAct->bOpCode < NUM_ACTION_CODES);
    result = sgActionFcns[pAct->bOpCode](pAct);
    if (result) {
      pTrigger->t.bCurrAction++;
    }
} « end trigger_execute »
```
What are triggers? /6

- Classic (visual) trigger editor (SCMDraft 2.0 – by Henrik Arlinghaus)

- Note the large values:
  - UnitID
  - Death table index
  - Etc.
What are triggers? /7

- Text trigger editor

- A private build of SCMDraft shows the EUD overflow addresses
The buffer overflow /1

- The buffer overflow bug in question is found in the “Extended Unit Death” trigger code:
  - The death_count() trigger condition
    - → Read anywhere primitive
  - The set/add/sub_death_count() trigger action
    - → Write anywhere primitive

- Triggers are read as-is from the chunk file and stored in a doubly-linked list:

```c
static BOOL CALLBACK maphdr_TRIG(HCHUNK hChunk, DWORD dwSize, LPARAM data) {
    if (dwSize % sizeof(TRIGGER)) return false;
    PTRIGGER pTrigBuf = (PTRIGGER)ALLOCTEMP(dwSize);
    if (!ReadChunk(hChunk, pTrigBuf)) {
        FREE(pTrigBuf);
        return false;
    }

    PTRIGGER pTrigger = pTrigBuf;
    int nTriggers = dwSize / sizeof(TRIGGER);

    for (int n = 0; n < nTriggers; n++, pTrigger++) {
        if (!AddTrigger(pTrigger))
            break;
    }

    FREE(pTrigBuf);
    return true;
}
```
The buffer overflow /2

- A death condition with out-of-bounds unit type (wType) or player number (dwPlayer) causes the read anywhere primitive
The buffer overflow /3

- A set death action causes a write anywhere and provides the following primitives:
  - \([\text{mem}] += l\text{Quantity}\)
  - \([\text{mem}] -= l\text{Quantity}\)
  - \([\text{mem}] = l\text{Quantity}\)

```c
static int act_set_deaths(PACTION pAct) {
    app_assert(pAct->bQualifier < NUM_QUALIFIER_CODES);
    switch (pAct->bQualifier) {
    case QUAL_SET:
        set_deaths(pAct->dwPlayer, pAct->wType, pAct->lParm);
        break;

    case QUAL_ADD_TO:
        add_deaths(pAct->dwPlayer, pAct->wType, pAct->lParm);
        break;

    case QUAL_SUB_FROM:
        sub_deaths(pAct->dwPlayer, pAct->wType, pAct->lParm);
        break;
    }
    return 1;
}
```

```c
static ULONG set_deaths(DWORD dwPlayer, UTYPE wType, ULONG lQuantity) {
    app_assert(dwPlayer < NUM_PLAYER_CODES);
    switch (dwPlayer) {
    case PLYR_NAVY: return nava_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_FOES: return foes_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_ALLIES: return allies_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_NEUTRALS: return neutrals_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_ALL: return all_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_GROUP_A: return group_A_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_GROUP_B: return group_B_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_GROUP_C: return group_C_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_GROUP_D: return group_D_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_UNUSED1: return group_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_UNUSED2: return group_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_UNUSED3: return group_enum(dwPlayer, wType, lQuantity, set_deaths);
    case PLYR_UNUSED4: return 0;
    case PLYR_THIS: dwPlayer = TriggerPlayer();
                    break;
    default:
        // dwPlayer is not a special code -- advance to next switch
        break;
    } // end switch dwPlayer 

    // dwPlayer is valid -- give resource to specified player
    if (dwPlayer >= NET_MAX_NODES) return 0;
    switch (wType) {
    case UNITS_BLOGS:
        s.g1GameCounts[COU_LOST_BLOGS][dwPlayer] -= lQuantity;
        break;

    case UNITS_FACTORYS:
        s.g1GameCounts[COU_LOST_FACTORYS][dwPlayer] -= lQuantity;
        break;
    default:
        s.g1UnitCounts[COU_UNIT_DEATH][wType][dwPlayer] -= lQuantity;
        break;
    }
    return 0;
} /* end set_deaths */
```
The buffer overflow /4

- An example of EUD triggers found inside an EUD map:

```c
// Trigger("All players")
{ // Trigger: 107
    Conditions:
    Deaths("Player 11", "Terran Marine", At least, 134217728);## Addr = 0x0058A38C; Value = 0x00000000

    Actions:
    Set Deaths("Player 11", "Terran Marine", Subtract, 134217728);## Addr = 0x0058A38C; Value = 0x00000000
    Set Deaths("Current Player", "Terran Marine", Add, 134217728);## Addr = 0x0058A398; Value = 0x00000000
}

// Trigger("All players")
{ // Trigger: 108
    Conditions:
    Deaths("Player 11", "Terran Marine", At least, 67108864);## Addr = 0x0058A38C; Value = 0x00000000

    Actions:
    Set Deaths("Player 11", "Terran Marine", Subtract, 67108864);## Addr = 0x0058A38C; Value = 0x00000000
    Set Deaths("Current Player", "Terran Marine", Add, 67108864);## Addr = 0x0058A398; Value = 0x00000000
}

// Trigger("All players")
{ // Trigger: 109
    Conditions:
    Deaths("Player 11", "Terran Marine", At least, 33554432);## Addr = 0x0058A38C; Value = 0x00000000

    Actions:
    Set Deaths("Player 11", "Terran Marine", Subtract, 33554432);## Addr = 0x0058A38C; Value = 0x00000000
    Set Deaths("Current Player", "Terran Marine", Add, 33554432);## Addr = 0x0058A398; Value = 0x00000000
}
```
EUD map emulation – Problem statement

• Given a StarCraft map that contains malformed input that triggers a read/write anywhere:
  • Is there a way to emulate the buffer overflow in a newer game version where:
    • The buffer overflow bug is fixed
    • Some addresses no longer exist in the new game version
    • Some addresses refer to new/different data structure format

• Can the emulator work on different architectures and operating systems?
Three steps solution

1. Identify
   • Identify / trace all the addresses used by an EUD map
   • Build a table of the addresses and identify what they represent in the game source code

2. Intercept
   • Intercept all out-of-bounds access
   • Redirect access using a translation table
     • Old address → New address

3. Emulate
   1. Missing memory addresses should be handled by code
   2. Dangerous memory changes should be filtered / changed accordingly (pointers, function callbacks, etc.)
Implementation challenges

1. Identify
   • Unfortunately, we did not have private or public symbols for StarCraft 1.16.1. I had to start reversing the game executable from scratch
   • How can I tell what addresses the maps are accessing?
   • What is the goal/intent behind a memory access?

2. Intercept
   1. No problems here. Luckily, we can funnel all the out-of-bounds read/writes to the emulation layer

3. Emulate
   1. Handle basic memory access emulation
   2. Emulate addresses that are no longer present
   3. Emulate incompatible structure types
Identify – Reversing the game /1

1. Reverse engineering efforts were impeded by the lack of debugging symbols:
   • Reverse engineered the game client from scratch
   • Used the closest source code snapshot for 1.16.1
   • Found the right compiler (VS 2003) and the approximate optimization switches
     ➢ Now I have debugging symbols for a binary that is very close to the public build

2. I used binary differencing plugins for IDA Pro
   1. PatchDiff2 - Tenable Network Security, Inc
   2. Diaphora - http://diaphora.re/
Identify – Reversing the game /2

• Binary differencing was limited:
  • Mismatched functions between the differenced binaries
  • Global variables were not identified
  • Optimized code and inlined functions made differencing harder

• Resorted to manual reverse engineering to bridge the limitations from BinDiffing

• Used scripting to automate the reversing task
  • Lots of IDAPython scripting was involved
Source code vs Disassembly view
Source code vs Hex-Rays pseudo-code

```c
static void trigger_execute(PTRIGGERNODE pTrigger) {

  // if this is a trigger action, there may be a delayed victory while
  if (pTrigger->t.1Flags & TF_VICTORY)
    sgVictoryCodes[sqWithTriggerPlayer] = CODE_DELAYED_VICTORY;

  // execute trigger actions until an action doesn't complete
  int result = 1;
  while (result & (pTrigger->t.bCurrAction < MAX_ACTIONS)) {
    pAct = pTrigger->t.tActions[pTrigger->t.bCurrAction];

    // the map editor can be used to disable actions
    if (pAct->iFlags & AF_DISABLED) {
      // Skip this action
      pTrigger->t.bCurrAction++;
      continue;
    }

    // no action indicates the end of the list
    if (pAct->iOpCode == ACT_NONE) {
      pTrigger->t.bCurrAction = MAX_ACTIONS;
      break;
    }

    // call the action function
    app_assert(pAct->iOpCode < MAP_ACTION_CODES);
    result = sgActionCodes[pAct->iOpCode](pAct);
    if (result) {
      pTrigger->t.bCurrAction++;
    }
  }

  // are we done with this trigger?
  if (pTrigger->t.bCurrAction < MAX_ACTIONS)
    return;

  // if this trigger paused the game, but never unpause, we want to
  if (pTrigger->t.1Flags & TF_PAUSED)
    sgActionFns[ACT_UNPAUSE_GAME](NULL);

  // reset the trigger appropriately
  if (pTrigger->t.1Flags & TF_KEEP_TRIGGER) {
    pTrigger->t.1Flags = 0;
  }

  // fgmough 1.0s begin
  pTrigger->t.1Flags &= ~(TF_EXECUTING | TF_ABORTED | TF_NO_ABORT);
  // fgmough 1.0s end
  else {
    pTrigger->t.1Flags = TF_COMPLETE;
  }
}
```

```c
void _usercall trigger_execute(z_TRIGGERNODE *pTrigger<esi>)
{
  unsigned _int32 trig_flag; // eax
  int ok; // eax
  unsigned _int8 bCurAction; // dl
  int iAction; // ecx
  bool b_disabled; // zf
  _action_t act; // ecx
  unsigned _int8 bOpCode; // al
  unsigned int fi; // eax

  trig_flag = pTrigger->t.1Flags | TF_EXECUTING;
  pTrigger->t.1Flags = trig_flag;
  if (trig_flag & TF_VICTORY)
    sgVictoryCodes[sqWithTriggerPlayer] = CODE_DELAYED_VICTORY;

  ok = 1;
  do {
    bCurAction = pTrigger->t.bCurrAction;
    if (bCurAction > (unsigned _int8)MAX_ACTIONS)
      break;

    // iAction = bCurAction;
    b_disabled = (pTrigger->t.tActions[iAction].bFlags & AF_DISABLED) == offsetof(z_TRIGGERNODE, m_link);
    act = &pTrigger->t.tActions[iAction];
    if (b_disabled)
      bOpCode = act->bOpCode;
    if (1bOpCode)
      {
        pTrigger->t.bCurrAction = MAX_ACTIONS;
        break;
      }
    ok = sgActionFns[bOpCode]();
    if (1ok)
      break;
    ++pTrigger->t.bCurrAction;
  } while (ok);

  pTrigger->t.bCurrAction = bCurAction + 1;
}
```
Automating data structure recovery
StarCraft Remastered collects game telemetry (including map information, etc.)

As of October 2017, we had around ~603,773 total unique maps played
  • Of which 17,916 were EUD maps (i.e. contained out of bounds indices)

After I managed to reverse engineer enough of the game, I wrote a tool to process all the maps, identify EUD maps and dump the out-of-bounds EUD addresses

Identify – Statically identify all addresses

Usage:
  --use-cache Do not parse again, just read the cache file.
  --input map_file Specify the input map file
  --workdir Specify the work directory where caches and report files will be generated
  --hidden-triggers [hexofs] Specify the hidden trigger offset within the strings table

s:\All-EUD-Maps> EUDump.exe
Identify – Statically identify all addresses /2
After aggregating the unique EUD addresses across all of the 17k EUD maps, I ended up with around ~800 variables used by popular EUD maps.

I wrote an IDAPython script to emit a table for all the unique addresses, their names and sizes.
Identify – Statically identify all addresses /4

• Static address discovery was not enough:
  • Some EUD maps were dereferencing pointers and reaching into the heap
  • Some structures are complicated and linked to other structures (linked lists, TCtrl*, TDialog*, etc.)

• Need more tools:
  • I realized the need for a dynamic EUD address tracer
  • I also needed a way to single step / debug triggers

• I developed an EUDTracer, a DLL that hooks the game and instruments all the relevant trigger handling code
Identify – Dynamic tracer /1

- The instrumented game binary calls into the tracer DLL upon each read/write
Identify – Dynamic tracer /2

• The Python table containing EUD addresses is passed to a source code generator to emit C code and tables

• The tracer uses that table to account for memory access
Identify – Dynamic tracer /3

- When the game loads an EUD map, the tracer DLL intercepts all out-of-bounds access.
- Any unknown address triggers a breakpoint for further analysis and identification.
- After I identify an unknown address, I add it to the Python table which is used to update the tracer’s EUD items table.

```c
void __stdcall eud_act_set_deaths(
    uint32_t val,
    uint32_t idx)
{
    inc_perf_counter(CT_SET_DEATH);
    DWORD addr = MAKE_EUD_ADDR_IDX(idx);
    auto eud = eud_get_item(addr, val, 1);
    if (eud == nullptr)
    {
        DBG_OUT("<Addr %08x: set_deaths; val: %08X>\n", addr, val);
        BPT_ONCE;
    }
}

void __stdcall eud_cond_deaths(
    uint32_t val,
    uint32_t idx)
{
    inc_perf_counter(CT_COND_DEATHS);
    DWORD addr = MAKE_EUD_ADDR_IDX(idx);
    auto eud = eud_get_item(addr, val, -2);
    if (eud == nullptr)
    {
        DBG_OUT("<Addr %08x: cond_deaths; val: %08X>\n", addr, val);
        BPT_ONCE;
    }
}
```
The tracer’s main role is to guarantee that all the addresses referred to from the EUD map are accounted for.
Identify – More debugging tools

• Having a way to record all accessed EUD addresses was not enough to understand the intent behind the access

• I had no real way to debug an EUD map:
  • I needed a way to nicely represent an EUD address
  • I needed to single step after each trigger
  • I needed a way to convert a series of read/write primitives to pseudo-code
If I wanted to trace triggers, I needed to have a way to convert an address to a nice variable representation.

So what is the symbolic representation of:

- \( 0x5187E8 + (0xC \times 3) + 4? \)
- \( gCards[3].pBtns \)

Identify – EUD address to symbolic name /1
With the help of the Hex-Rays decompiler and other metadata, I wrote the function “R” to resolve an address into a nice symbolic name.

If the array’s indices are based on enums, then “R” will properly show the enum name instead of a numeric index.
Identify – Static pseudocode generator / 1

• SCMDraft trigger editor textually represents the trigger script:

```
Trigger(" || Team 1 ||", " || Team 2 ||") { // Trigger: 1022

Conditions:
Deaths("Current Player", "Right Wall Flame Trap", Exactly, 1);## Addr = 0x0058CB88; Value = 0x00000001
Deaths("Current Player", "Right Upper Level Door", Exactly, 2);## Addr = 0x0058CA38; Value = 0x00000002
Deaths("Current Player", "Mineral Field (Type 1)", Exactly, 11);## Addr = 0x0058C498; Value = 0x0000000B
Bring("Current Player", "Terran Science Vessel", "Invalid Location", At least, 1);
Bring("Current Player", "Terran Science Vessel", "Invalid String", At least, 1);

Actions:
Set Deaths("Current Player", "Floor Missile Trap", Set To, 0);## Addr = 0x0058C9A8; Value = 0x00000000
Set Deaths("Current Player", "Right Wall Flame Trap", Set To, 100);## Addr = 0x0058CB88; Value = 0x00000064
Set Deaths("Current Player", "Right Wall Missile Trap", Subtract, 25);## Addr = 0x0058CB58; Value = 0x00000019
Set Deaths("Current Player", "Terran Valkyrie", Add, 50);## Addr = 0x0058A678; Value = 0x00000032
Set Deaths("Current Player", "Right Upper Level Door", Set To, 0);## Addr = 0x0058CA38; Value = 0x00000000
Set Deaths("Current Player", "Protost Shield Battery", Set To, 144);## Addr = 0x0058C3D8; Value = 0x00000090
Move Location("Current Player", "Terran Science Vessel", "Invalid Location", "Invalid Location");
Remove Unit("Current Player", "Terran Science Vessel");
Set Deaths("Player 7", "Protost Arbiter", Set To, 12);## Addr = 0x0058B0CC; Value = 0x0000000C
Set Deaths("Player 12", "Int:18768", Add, 20905984);## Addr = 0x00666290; Value = 0x013F0000
Set Deaths("Player 4", "Int:27270", Add, 1473);## Addr = 0x006C59C9; Value = 0x0000005C
Set Deaths("Player 2", "Int:27284", Add, 1287);## Addr = 0x006C9F28; Value = 0x00000057
Set Deaths("Player 11", "Int:27278", Add, 40);## Addr = 0x006C9E2C; Value = 0x00000028
Create Unit with Properties("Current Player", "Yggdrasill (Overlord)", 1, "Invalid String", 3);
Set Deaths("Player 12", "Int:18768", Subtract, 20905984);## Addr = 0x00666290; Value = 0x013F0000
Set Deaths("Player 4", "Int:27270", Subtract, 1473);## Addr = 0x006C59C9; Value = 0x0000005C
Set Deaths("Player 2", "Int:27284", Subtract, 1287);## Addr = 0x006C9F28; Value = 0x00000057
Set Deaths("Player 11", "Int:27278", Subtract, 40);## Addr = 0x006C9E2C; Value = 0x00000028
Set Deaths("Current Player", "Right Wall Flame Trap", Set To, 0);## Addr = 0x0058CB88; Value = 0x00000000
Play WAV("sound\Bullet\zdeAtt00.wav", 0);
Comment("ząÅ¥%ÅÄÅ");
Preserve Trigger();
}
``
I wrote a converter from the triggers text to C pseudo-code (convert triggers to an AST and then emit as C pseudo-code).
Identify – Static pseudocode generator /3

- Trigger text converted to C pseudo-code (trig2cpp()):

```c
void trigger_1022()
{
    if (!(_g_s.gIUnitCounts[COU_UNI_DEATH][UNI_STARTLOC][PLYR_TWO] == 0x00000001) &&
        (!(_g_s.gIUnitCounts[COU_UNI_DEATH][UNI_INSTALL_SPIKED_DOOR1][PLYR_TWO] == 0x00000002) &&
         (_g_s.gIUnitCounts[COU_UNI_DEATH][UNI_MINE_V2][PLYR_TWO] == 0x00000008)) &&
        (Bring("Current Player", "Terran Science Vessel", "Invalid Location", At least, 0x00000001)) &&
        (Bring("Current Player", "Terran Science Vessel", "Invalid String", At least, 0x00000001)))
    {
        _g_s.gIUnitCounts[COU_UNI_DEATH][UNI_INSTALL_HATCH][PLYR_TWO] = 0x00000000; // 0058c9a8
        _g_s.gIUnitCounts[COU_UNI_DEATH][UNI_STARTLOC][PLYR_TWO] = 0x00000004; // 0058cb88
        _g_s.gIUnitCounts[COU_UNI_DEATH][UNI_INSTALL_WALL_FLAME][PLYR_TWO] = 0x00000019; // 0058cb58
        _g_s.gIUnitCounts[COU_UNI_DEATH][UNI_Z_COCOON][PLYR_TWO] = 0x00000032; // 0058ae78
        _g_s.gIUnitCounts[COU_UNI_DEATH][UNI_INSTALL_SPIKED_DOOR1][PLYR_TWO] = 0x00000000; // 0058ca38
        _g_s.gIUnitCounts[COU_UNI_DEATH][UNI_P_KHAY_FORMATION][PLYR_TWO] = 0x00000090; // 0058c3d8
        Move Location("Current Player", "Terran Science Vessel", "Invalid Location", "Invalid Location");
        Remove Unit("Current Player", "Terran Science Vessel");
        _g_s.gIUnitCounts[COU_UNI_DEATH][UNI_P_ARBITER][PLYR_SEVEN] = 0x0000000C; // 0058b0cc
        guwSpriteImage[SPR_Z_MUTALID_DEATH] = 0x013F0000; // 00666290
        gfxFlingyAccel[FLI_Z_OVERLORD] = 0x00000051; // 006c9c90
        gfxFlingyMaxVel[FLI_Z_OVERLORD] = 0x00000050; // 006c9f28
        gubFlingyMaxTurn[FLI_Z_OVERLORD] = 0x00000028; // 006c9e2c
        Create Unit with Properties("Current Player", "Yggdrasill (Overlord)", 1, "Invalid String", 0x00000003);
        guwSpriteImage[SPR_Z_MUTALID_DEATH] = 0x013F0000; // 00666290
        gfxFlingyAccel[FLI_Z_OVERLORD] = 0x00000051; // 006c9c90
        gfxFlingyMaxVel[FLI_Z_OVERLORD] = 0x00000050; // 006c9f28
        gubFlingyMaxTurn[FLI_Z_OVERLORD] = 0x00000028; // 006c9e2c
        _g_s.gIUnitCounts[COU_UNI_DEATH][UNI_STARTLOC][PLYR_TWO] = 0x00000000; // 0058cb88
        Play WAV("sound\Bullet\zdeAtt08.wav", 0x00000000);
        Comment("±½%ÀèÆ");
        Preserve Trigger();
    }
} « end trigger_1022 »
```
With IDA’s conditional breakpoints and the Appcall feature, I wrote a dynamic pseudocode generator:

- It helps debug the map trigger logic during runtime
- It helps in the discovery and understanding of dynamic triggers (generated by the EUD compiler from \textit{trgk})

- Conditional breakpoints are set at strategic entrypoints (pre, in and post trigger execution)
Identify – Dynamic pseudocode generator /2

- Conditional breakpoints dynamically build the AST on access
• The debug script has a ‘Single step’ switch to break after each trigger
• Pseudocode is emitted on the fly
The “Single step” switch can be configured to print the pseudocode on the fly as the map triggers executes without suspending the game.

Demo – Dynamic pseudocode generator /2
In the first step (identify):
1. We built all the required static and dynamic tracers
2. We created the EUD table with all known addresses and their symbolic names
3. We have enough tools to identify any address and trace where it came from

Now we need to intercept the out-of-bounds access in the new code base
**Intercept /2**

### Read primitives interception

```
switch (wType)
{
  case UNITS_ALL:
    return bOverflow ? 0 : s->glGameCounts[COU_LOST_MEN][dwPlayer] - s->glGameCounts[COU_KILLED_MEN][dwPlayer];
  case UNITS_MEN:
    return bOverflow ? 0 : s->glGameCounts[COU_LOST_MEN][dwPlayer];
  case UNITS_BLODS:
    return bOverflow ? 0 : s->glGameCounts[COU_LOST_BLODS][dwPlayer];
  case UNITS_FACTORIES:
    return bOverflow ? 0 : s->glGameCounts[COU_LOST_FACTORIES][dwPlayer];
  default:
    if (h_eud_is_eud_map)
    {
      auto pCond = (PCONDITION)1Param;
      return eud_cmd_deaths;
      dwPlayer,
      wType,
      pCond);
    }
    assert(wType < NUM_UNITS);  
    return bOverflow ? 0 : s->glUnitCounts[COU_UNIT_DEAD][wType][dwPlayer];
}
```

### Write primitives interception

```
default:
  if (h_eud_is_eud_map)
  {
    eud_set_deaths(dwPlayer, wType, 1Quantity, 0);
    return 0;
  }
  if (bOverflow || wType >= NUM_UNITS)
  return 0;
  s->glUnitCounts[COU_UNIT_DEAD][wType][dwPlayer] = 1Quantity;
  break;
}
```

```
From the emulator’s perspective, all EUD map logic boils down to two actions:

1. Read anywhere \( \rightarrow \) value = read_vmem(eud_addr)
2. Write anywhere \( \rightarrow \) write_vmem(eud_addr, value)
Emulate

In basic scenarios, the emulation is very simple:
1. Compute the full virtual address (EUD address) from the *dwPlayer* and *wType* indices
2. From the EUD address, find the equivalent new address (backing data) in the current game version
3. Compute the offset and read or write from/to the new address
Let’s extend the previous Python table and attach the source file name were each variable is located

The table defines: virtual address, item size, source file name, emulation flags, and backing variable name
Running the EUD table generation script patches the source code and exports all referenced variables:

Emulate – Variables mapping /2
Emulate – Variables mapping

Exported variables example:

```c
// data tables
UWORD guwFlingySprite[NUM_FLINGIES];
ULONG gxFlingyMaxVel[NUM_FLINGIES];
UWORD gxFlingyAccel[NUM_FLINGIES];
ULONG gxFlingySlow[NUM_FLINGIES];
UBYTE gubFlingyMaxTurn[NUM_FLINGIES];
UBYTE gubFlingyMinBank[NUM_FLINGIES];
UBYTE gubFlingyMoveType[NUM_FLINGIES];

/// EUD EXTERNS - AUTOGENERATE BEGIN /////
static_assert(sizeof(gubFlingyMoveType) == 0xd1, "EUD size mismatch for gubFlingyMoveType");
void *eud_ptr_gubFlingyMoveType = reinterpret_cast<void*>(&gubFlingyMoveType);
static_assert(sizeof(gxFlingySlow) == 0x344, "EUD size mismatch for gxFlingySlow");
void *eud_ptr_gxFlingySlow = reinterpret_cast<void*>(&gxFlingySlow);
static_assert(sizeof(gxFlingyAccel) == 0x1a2, "EUD size mismatch for gxFlingyAccel");
void *eud_ptr_gxFlingyAccel = reinterpret_cast<void*>(&gxFlingyAccel);
static_assert(sizeof(gubFlingyMaxTurn) == 0xd1, "EUD size mismatch for gubFlingyMaxTurn");
void *eud_ptr_gubFlingyMaxTurn = reinterpret_cast<void*>(&gubFlingyMaxTurn);
static_assert(sizeof(gxFlingyMaxVel) == 0x344, "EUD size mismatch for gxFlingyMaxVel");
void *eud_ptr_gxFlingyMaxVel = reinterpret_cast<void*>(&gxFlingyMaxVel);
static_assert(sizeof(gwuFlingySprite) == 0x1a2, "EUD size mismatch for guwFlingySprite");
void *eud_ptr_gwuFlingySprite = reinterpret_cast<void*>(&guwFlingySprite);
static_assert(sizeof(gubFlingyMinBank) == 0xd1, "EUD size mismatch for gubFlingyMinBank");
void *eud_ptr_gubFlingyMinBank = reinterpret_cast<void*>(&gubFlingyMinBank);
```
No need to make static variables global:

- The generator has an option that lets you pick a name for the exported variable

```c
// CImage.cpp

static CLists *sgpFreeHead;
static CLists *sgpFreeTail;

#if EUD_ENABLED
    static_assert(sizeof(sgpFreeHead) == 0x4, "EUD size mismatch for sgpFreeHead");
    void *eud_ptr_images_sgpFreeHead = reinterpret_cast<void*>(&sgpFreeHead);
    static_assert(sizeof(sgpFreeTail) == 0x4, "EUD size mismatch for sgpFreeTail");
    void *eud_ptr_images_sgpFreeTail = reinterpret_cast<void*>(&sgpFreeTail);
#endif // (EUD_ENABLED)

```
Emulate – The EUD table /1

• The “eud_table.cpp” is autogenerated from the Python table. It refers to all the exported variables from various source code files

• It is used to populate the emulator’s virtual memory layout

• Items also have associated flags that instruct the emulator which EUD adapter handles which address

• Note: the “g_nothing” variables are alignment bytes in SC 1.16.1. The map makers use that space for storing variables

• A “nullptr” backing data almost always indicates that the variable is to be handled purely by an adapter code
Emulate – The EUD table /2

• The “eudExtern.h” is autogenerated from the Python table

• It exposes all the known EUD variables
  • Very handy for accessing static variables from anywhere in the code when needed

```c
#include <cstdint>

#pragma once
extern void *eud_ptr_sgCard;
extern void *eud_ptr_sgnScrollRates;
extern void *eud_ptr_gbInMsgMode;
extern void *eud_ptr_sgszToPlayerPrompt;
extern void *eud_ptr_gGameHeader;
extern void *eud_ptr_gpIconsGrp;
extern void *eud_ptr_sgbSelectionUpdate;
extern void *eud_ptr_sgbStatPortUpdate;
extern void *eud_ptr_gubFlingyMoveType;
extern void *eud_ptr_gxFlingySlow;
extern void *eud_ptr_gxFlingyAccel;
extern void *eud_ptr_gubFlingyMaxTurn;
extern void *eud_ptr_gxFlingyMaxVel;
extern void *eud_ptr_gwFlingySprite;
extern void *eud_ptr_gubFlingyMinBank;
extern void *eud_ptr_sgnPrevPalId;
extern void *eud_ptr_sgpRpMap;
extern uint32_t eud_export_REPULSE_MAP_SIZE;
extern void *eud_ptr_g_ActiveNationID;
extern void *eud_ptr_g_LocalNationID;
```
Due to the nature of the overflow, the following restrictions apply:

- An EUD address is always 4 bytes aligned
- An EUD value is a 32bits integer
Emulator architecture /2

Shadow table
• It contains the needed memory contents from the SC 1.16.1 binary

Virtual memory
• It uses the address-to-handlers lookup table
• It maps an EUD address range to an EUD table entry → EUD handler/adapter

• The table entry for an EUD item describes:
  • The backing data (the new variable address, if present)
  • The flags which tell the emulator which EUD adapter (handler) to use for emulation
A specialized EUD adapter is needed when:
• Handling non-standard data types
• When dealing with EUD addresses that no longer map to anything in the new game client

The following 5 virtual methods are exposed
• read_vmem() → Return a 32bits value
• write_vmem() → Write a 32bits value
• backup() → Item specific backup code
• restore() → Item specific restore code
• deferred_write() → Invoked after all the triggers have executed. Gives a chance to batch process writes
EUD adapters – Basic /1

The basic EUD adapter *(eud_vmemitem_t class)* handles basic data types:
1. The emulator computes the full EUD address
2. Finds the new variable’s base address and converts the EUD address to an offset
3. The appropriate adapter is then called with the desired offset to read/write from/to

```c
bool eud_vmemitem_t::read_vmem(
    eud_addr_type addr,
    eud_value_type &value)
{
    eud_vmemitem_t *vitem = find_range(vmem, addr);
    if (vitem == nullptr)
    {
        set_not_supported(false);
        return false;
    }
    return vitem->read_vmem(
        this,
        addr - vitem->addr,
        value);
}
```

```c
void *eud_vmemitem_t::write_vmem(
    eud_addr_type addr,
    eud_value_type value)
{
    eud_vmemitem_t *vitem = find_range(vmem, addr);
    if (vitem == nullptr)
    {
        set_not_supported(true);
        return nullptr;
    }
    return vitem->write_vmem(
        this,
        addr - vitem->addr,
        value);
}
```

This simple translation approach works nicely for basic types
The basic (pass-thru) adapter is good for most cases:

- Byte, Word, Dword
- The emulator can cross boundaries between two items
- Basic types arrays are also supported

EUD adapters – Basic /2

Reading a value from the end involves reading from two different adapters (handlers)
Wait a minute, we need one more primitive!

- We covered two primitives:
  1. *mem asg_op = const
     - asg_op → +=, =, -=
  2. if (*mem cmp_op const) { actions ... }
     - cmp_op → ==, >=, <=

- How do we get the following primitive?
  - *mem1 asg_op *mem2

**Using binary search!**
The *a = *b primitive

- Trigger condition:
  1. Probes the value of src_var

- Trigger action:
  1. Increments the value of dst_var
  2. Decrement the value of src_var
  3. src_var’s value eventually reaches zero
  4. Backup changes into var_copy

The same primitive is repeated to copy var_copy back to dst_var

This primitive is expensive and generates lots of triggers

```c
void trigger_0()
{
    var_copy = 0;
    dst_var = 0;
}
void trigger_1()
{
    if ((src_var >= 0x00000000))
    {
        src_var -= 0x00000000;
        dst_var += 0x00000000;
        var_copy += 0x00000000;
    }
}
void trigger_2()
{
    if ((src_var >= 0x04000000))
    {
        src_var -= 0x04000000;
        dst_var += 0x04000000;
        var_copy += 0x04000000;
    }
}
void trigger_3()
{
    if ((src_var >= 0x02000000))
    {
        src_var -= 0x02000000;
        dst_var += 0x02000000;
        var_copy += 0x02000000;
    }
}
void trigger_...()
{
    if ((src_var >= 0x00000100))
    {
        src_var -= 0x00000100;
        dst_var += 0x00000100;
        var_copy += 0x00000100;
    }
}
void trigger_...()
{
    if ((src_var >= 0x00000004))
    {
        src_var -= 0x00000004;
        dst_var += 0x00000004;
        var_copy += 0x00000004;
    }
}
void trigger_...()
{
    if ((src_var >= 0x00000002))
    {
        src_var -= 0x00000002;
        dst_var += 0x00000002;
        var_copy += 0x00000002;
    }
}
void trigger_...()
{
    if ((src_var >= 0x00000001))
    {
        src_var -= 0x00000001;
        dst_var += 0x00000001;
        var_copy += 0x00000001;
    }
}
} /* end trigger_0 */
EUD adapters – Pointers /1

• Pointers are 32bits in SC 1.16.1

• Obviously, we cannot just use the pass-thru basic emulation
  • Pointers have to be translated from EUD virtual addresses to real addresses

• The primitive “*ptr1 = *ptr2” invoked from the EUD triggers will spoil the pointer value until the binary search is over
  • What to do with incomplete pointer values?
EUD adapters – Pointers /2

- Changes to a physical pointer value should not take effect unless the virtual pointer value passes a “pointer validity check function” → Does the virtual pointer have a proper real pointer equivalent?

- Rely on the shadow pointer value when working with incomplete virtual pointer values for future reads / writes:

```
void *game_ptr;
uint32_t game_ptr_shadow;
bool game_ptr_dirty;
```

```
uint32_t game_ptr;
```
EUD adapters – Pointers /3

• The `eud_object_ptr_adapter_t` is constructed with backing data pointing to a reference to a real pointer that we want to expose to the EUD emulator.

```cpp
template <class T>

class eud_object_ptr_adapter_t : public eud_vmemitem_t
{
protected:
    eud_addr_type shadow_ptr;

public:
    T **ptr() const { return *(T **)backing_data; }

    // Convert an physical pointer to an EUD pointer
    eud_value_type physical_to_eudaddr(eud_addr_t *EMU)
    {
        // Read the live value...
        auto obj = (* EMU)
        // ...and translate it to a virtual address
        return obj == nullptr ? 0 : EMU->get_object_vptr(obj);
    }

    virtual bool read_vmem(
      eud_addr_t *emu,
      eud_addr_type offs,
      eud_value_type &value) override
    {
        // When not dirty, read the live value
        if (!is_dirty())
        {
            shadow_ptr = physical_to_eudaddr(emu);
            value = shadow_ptr;
            return true;
        }
        // Update the shadow value
        set_pval( &shadow_ptr, value, q);

        // Allow nullptr assignment
        if (shadow_ptr == 0)
        {
            ptr() = nullptr;
        }
        else
        {
            // Update the real pointer only if it gets translated
            // from an EUD addr to a physical pointer
            if (unit != nullptr)
            {
                ptr() = unit;
                clear_dirty();
            }
        }
        return true;
    }

    virtual bool write_vmem(
        eud_addr_t *emu,
        eud_addr_type offs,
        eud_value_type value,
        int q) override
    {
        if (is_dirty())
        {
            shadow_ptr = get_vptr(emu);
            set_dirty();
        }
        // Update the shadow value
        set_pval( &shadow_ptr, value, q);

        // Allow nullptr assignment
        if (shadow_ptr == 0)
        {
            ptr() = nullptr;
        }
        else
        {
            // Update the real pointer only if it gets translated
            // from an EUD addr to a physical pointer
            if (unit != nullptr)
            {
                ptr() = unit;
                clear_dirty();
            }
        }
        return true;
    }
};
```
EUD adapters – Function pointers /1

- What about EUD logic that does function pointer arithmetic?
EUD adapters – Function pointers /2

- Pointer arithmetic make sense only in the EUD virtual memory addressing space

- For the real pointer addressing we have to translate to proper pointers and account for function prototype compatibility

- Basic implementation idea:
  1. \( vaddr += voffs \)
  2. \( paddr = \text{find\_real\_fptr}(vaddr, \text{function\_prototype\_id}) \)
  3. if (\( paddr \neq \text{nullptr} \)) \( \Rightarrow \text{struct.pFn} = paddr; \)

- In the emulator, such cases are handled with the \textit{eud\_struct\_with\_ptr\_adapter\_t}

Virtual function pointers and their prototypes table

```c
#define TButtons_function_pointers pfCanDisplay and pfBtnAction
{
    "src_file": "r\SWARIlang\stmtbtn.cpp", "group": "Card/Buttons function pointers",
    "addr": 0x004282d9, "size": 0x00000000, "ida_name": "bf_always", "name": "bf_always", "flags": "EIF_FUNC_PTR | EIF_SRC_TBUTTON"},
    {"addr": 0x00424440, "size": 0x00000004, "ida_name": "order_move", "name": "order_move", "flags": "EIF_FUNC_PTR | EIF_SRC_TBUTTON"},
    {"addr": 0x0042357a, "size": 0x00000000, "ida_name": "order_stop", "name": "order_stop", "flags": "EIF_FUNC_PTR | EIF_SRC_TBUTTON"},
    {"addr": 0x004269f0, "size": 0x00000004, "ida_name": "bf_can_attack", "name": "bf_can_attack", "flags": "EIF_FUNC_PTR | EIF_SRC_TBUTTON"},
    {"addr": 0x004243b0, "size": 0x00000004, "ida_name": "order_attack", "name": "order_attack", "flags": "EIF_FUNC_PTR | EIF_SRC_TBUTTON"},
    {"addr": 0x00424140, "size": 0x00000004, "ida_name": "order_petrol", "name": "order_petrol", "flags": "EIF_FUNC_PTR | EIF_SRC_TBUTTON"},
    {"addr": 0x00423770, "size": 0x00000004, "ida_name": "order_hold_pos", "name": "order_hold_pos", "flags": "EIF_FUNC_PTR | EIF_SRC_TBUTTON"},
```
EUD adapters – Incompatible structures /1

- Various data structures have changed between SC 1.16.1 and SC:R

- Pass-thru adapters are not helpful in this case

```
struct eud_CUnit
{
    int unit_id;    /* 0x00 */
    char unit_name[80]; /* 0x04 */
    eud_CUnit *linked_unit; /* 0x54 */
    eud_CImage *linked_Sprite; /* 0x58 */
}
```

```
struct CUnit
{
    char unit_name[80];    // 0x00
    int field1;            // 0x05
    int field2;            // 0x05
    int unit_id;           // 0x05
    eud_CUnit *linked_unit; // 0x05
    eud_CImage *linked_Sprite; // 0x05
}
```

- A specialized adapter is needed to convert between both structures:
  - **Read operation**: translates from physical structure to virtual structure
  - **Write operation**: translates from virtual structure to physical structure
EUD adapters – Incompatible structures /2
EUD adapters – Linked lists

- In SC 1.16.1
  - Triggers were stored in a **Storm** linked list data structure
  - **Storm** is a library that provides containers and platform independent functionality

- In SC:R
  - Triggers are stored as `blz::list<_trigger>`
  - ‘blz’ is the equivalent of STL’s std namespace

- Other structures in the old game also use **Storm** lists while the new game uses different containers
EUD adapters – Triggers /1

Because triggers are hard to program, the South Korean hacker (nicknamed Trigger King / trgk) wrote a trigger compiler:

1. You write proper logic in a JavaScript/Python like language called **epScript**

2. The **epScript** gets compiled into a bunch of triggers and is then injected into the appropriate map chunks

3. Map containing triggers compiled with **epScript** can be identified using the bootstrap code that links regular triggers into the dynamic triggers (inside the strings table)
EUD adapters – Triggers /2

- epScript is a very powerful language:
  - The Mario Exodus EUD map was written in that language

- Its compiler hides additional triggers in the cave area of the strings chunk:
  - Making it hard to reverse-engineer compiled triggers
  - One needs to write a triggers decompiler to recover the logic

- Compiled triggers are self-modifying and very optimized:
  - Loops, function calls and other control flow related functionality are implemented using self-modifying triggers that change the trigger node links (next and prev links)
EUD adapters – Triggers /3

• EUD maps locate the pointer to the string table (gpMapStr) and adds a constant offset pointing to the additional dynamic triggers inside the string table (see slide 17)

• EUD maps then patch the \texttt{m\_prevlink} and \texttt{m\_next} links as needed to introduce as many triggers as needed
  • Inserting new triggers dynamically was never supported in StarCraft. Only the EUD emulator allows such activity.

• Compiled/dynamic triggers are the basis of complex and elaborate EUD maps
  • Therefore, supporting dynamic triggers was the first thing added to the EUD emulator
EUD adapters – Triggers /4

• From the emulator’s perspective, there are two kinds of triggers:
  • Initial triggers originating from the triggers chunk
  • Dynamic triggers linked to the triggers list by patching their node links

• When StarCraft needs to execute triggers after each game loop:
  • The emulator knows how to serve both static triggers and dynamic EUD triggers
  • The emulator does not replicate the backing data (the trigger node data) whenever possible

---

SC:R → blz::list< trigger> :

```
_trigger0 _trigger1 ... _triggerN
```

SC1.16: stormlist< trigger> :

```
_trigger0 _trigger1 ... _triggerN
shadow: prev|next shadow: prev|next shadow: prev|next
```

String table:

```
Strings chunk data
Actual string table (TStrTbl) Extra chunk data: dynamic triggers
```

(Dynamic triggers inserted at the end of the strings table)
The Storm node EUD adapter hosts the node links as shadow variables.

```cpp
//
// template <class T>
// class eud_storm_node_adapter_t: public eud_vmemitem_t
// {
//
//     template <class TT>
//     friend class eud_storm_list_adapter_t;
//
//     private:
//         eud_STORM_TSLink shadow_link;
//
//     // Returns nullptr if offset is outside the shadow link bounds, the caller
//     // then knows how to read the node data in that case
//     eud_value_type get_pval(eud_addr_type offs)
//     {
//         if (offs == offsetof(eud_STORM_TSLink, m_prevlink))
//             return &shadow_link.m_prevlink;
//         else if (offs == offsetof(eud_STORM_TSLink, m_next))
//             return &shadow_link.m_next;
//         else
//             return nullptr;
//     }
//
// public:
//
//     enum {
//         NODE_SIZE = sizeof(eud_STORM_TSLink) + sizeof(T)
//     };
//
// static eud_storm_node_adapter_t *create(T *data)
// {
//     auto vitem = new eud_storm_node_adapter_t();
//     vitem->addr = emu->reserve_addr(NODE_SIZE);
//     vitem->flags = EIF_DYNAMIC | EIF_IS_STORM_LIST_NODE;
//     vitem->size = NODE_SIZE;
//     vitem->backing_data = data;
//     emu->set_item(vitem);
//     return vitem;
// }
//
// virtual bool write_vmem(eud_emu_t *emu,
//     eud_addr_type offs,
//     eud_value_type value,
//     int q = 0) override
// {
//     // Accessing node link structure?
//     eud_addr_type *pval = get_pval(offs);
//     if (pval != nullptr)
//     {
//         // Update node structure
//         set_pval(pval, value, q);
//         return true;
//     }
//     return eud_vmemitem_t::write_vmem(emu,
//         offs - sizeof(shadow_link),
//         value,
//         q);
// }
//
// virtual bool read_vmem(eud_emu_t *emu,
//     eud_addr_type offs,
//     eud_value_type &value) override
// {
//     eud_addr_type *pval = get_pval(offs);
//     if (pval != nullptr)
//     {
//         value = *pval;
//         return true;
//     }
//     return eud_vmemitem_t::read_vmem(emu,
//         offs - sizeof(shadow_link),
//         value);
// }
//
// ```
The Storm list adapter implements an STL compatible iterator.

From the iterator’s perspective, any node pointers outside the list have their node links and data in the virtual memory.

```cpp
class iterator {
public:
    iterator &operator++() {
        eud_addr_type next = m_cur + offsetof(eud_STORM_TSLink, m_next);
        // Return a terminal iterator if it is not possible to read the next link
        // or the link is negative (terminal by nature)
        if (!container->emu->read_vmem(next, m_cur) || finished())
            *this = container->end();
        return *this;
    }

    iterator operator++(int) {
        iterator tmp = *this;
        ++*this;
        return tmp;
    }

    reference operator*() {
        // Find the item hosting that address
        eud_addr_type *vitem = container->emu->find_item(m_cur);
        if (vitem == nullptr)
            EUD_ASSERT(("Failed to dereference storm list iterator!", false));
        static auto empty_node = value_type();
        return empty_node;
    }

    // Is that an adapted trigger node?
    eud_addr_type offs;
    if (((vitem)->flags & EUP_IS_STORM_LIST_NODE) != 0) {
        // The backing data is used as-is
        offs = 0;
    } else {
        // This trigger node exists in arbitrary memory,
        // Use the backing data and the current node as offset
        offs = (m_cur - vitem->addr + sizeof(eud_STORM_TSLink));
    }
    return *pointer((char *)vitem->backing_data + offs);
} // end pointer

pointer operator->() {
    return &**this;
}
```
EUD adapters – Partial buffers

• Partial buffers adapters are used whenever the virtual item size is greater than the physical item size:

  SC 1.16.1 item (virtual): data

  SC:R item (physical): smaller data unmapped

• The adapter serves the mapped data when the access offset is within the mapped range

• It will serve zeros w/o failing when the unmapped area is accessed
EUD adapters – Deferred writes

1. Certain adapters resort to using deferred writes as means to speed-up the emulation

2. The EUD map writes in chunks of 4 bytes at a time
   - We don’t want to re-construct real game data while the EUD map is still writing the changes

3. Instead, a write handler simply passes-thru the writes to a temporary buffer and marks the adapter as dirty
   - (Reads from dirty offsets are served from the temporary buffer for consistency)

4. After all triggers are executed in that game loop, the emulator invokes all the dirty adapters’ deferred write callbacks

5. Inside the deferred write callback, the temporary buffer is then used to reconstruct the real structures used by the game. The adapter dirty flag is then cleared.
EUD adapters – Deferred writes /2

Deferred write example adapter:

1. The status text adapter lets the EUD maps write to a temporary buffer

2. Afterwards, the adapter reconstructs the proper status text structures that are compatible with the new game (SC:R) code
EUD adapters – Bounded array elements

- Various game data variables are integer arrays

- Sometimes, the elements in the array must have bounded values
  - Naturally, the pass-thru (basic) adapter is not suitable (because no validation takes place)

- The bounded array adapter also leverage a shadow array table for all the elements that have incomplete / invalid values

- Only after the written values are valid (within the specified bounds) then changes are reflected into the backing data
• The Unit Flingy array’s values have an upper bound of 209
EUD adapters – Full adapters list /1

Throughout the creation of the EUD emulator, various adapters were devised whenever a new problem is encountered:

- **eud_adapter_cards**
  - Supports total customization of units command cards

- **eud_adapter_csprites** and **eud_adapter_cunit**
  - Allows controlled modifications into the CSprite and CUnit structures

- **eud_adapter_group**
  - Allows bitmap shuffling inside certain game animation frames

- **eud_adapter_keytable**
  - Allows EUD maps to intercept key presses (‘a’, ‘s’, ‘w’, ‘d’, key up and key down for example)
EUD adapters – Full adapters list /2

- **eud_adapter_mpq**
  - Allows support for protected maps.
  - Refer to MPQ frozen maps: [https://github.com/phu54321/euddraft/tree/master/freeze](https://github.com/phu54321/euddraft/tree/master/freeze)

- **eud_adapter_msgtbl**
  - Read access into the in-game chat messages ("Chatting War" EUD maps)

- **eud_adapter_partial_buffer**
  - Various non-emulated or no longer existent variables are handled with this adapter

- **eud_adapter_playerdata**
  - Lets EUD maps read player information (name, race, color, etc.)
EUD adapters – Full adapters list /3

- **eud_adapter_pointers**
  - All pointer related adaption code
  - Supports partial pointers (backed by shadow values)

- **eud_adapter_stattxt**
  - Unit status text and hotkeys manipulation

- **eud_adapter_stormlist**
  - Allows high-level emulation of Storm lists

- **eud_adapter_structwithptr**
  - Used to emulate structures that contain a mix of basic types (pass-thru) and pointers (incomplete pointers + virtual <-> physical conversion)

- **eud_adapter_triggers**
  - Supports dynamic triggers emulation
Questions?