Dark Souls II P2P Networking Exploits and Vulnerabilities

Contents

Preface

This document focuses on PC cheating and exploits in Dark Souls II (ver. 1.02 : cal. 2.02).

Similar to the other games in the series, Dark Souls II has a serious cheater problem. This game was the first Dark Souls game to introduce an anti-cheat, and this has been rather successful in ensuring that players who cheat are punished. While this has been effective at the start of the game's lifecycle, cheaters have worked out what flags with the anticheat and what does not. The anticheat is now being abused by cheaters to get innocent players banned by corrupting their player data with modified game packets.

I've based this document from the exploits I have found while developing an anti-cheat tool for Dark Souls II (known as "Blue Acolyte").

This document will examine each p2p packet "ID" (the number value assigned to the p2p packets in the in-game code) and document vulnerabilities and exploits that myself and a few others have found throughout the past few years. Where possible, I have tried to label the function of the packet, however without access to the source code I cannot reliably tell what each packet's true function is.

I have also only listed exploits that I know of, and feel like should be addressed ASAP – Minor or harmless ones have been omitted.

Receivers

NetSummonPacketCtrl

6 - ”Register steam user”

Below is how I understand that this packet data is structured:

```
struct SPacket6 {
	char pSteamIDEntry[17][5];
	uint32_t iSteamIDCount;
};
```

This packet is sent to a connecting player, and contains a list of 5 steamid64 entries formatted as unicode strings (1 per active player in the session).

Buffer overflow

The issue with this packet is with the entry `”iSteamIDCount”. Although the packet is a set size and can only hold five 16 character strings (plus a null-terminate character), this value is used to tell the parsing function (sub_) how many entries are declared in the packet.

A malicious player can simply set this value to MAX_UINT32 and cause buffer overflow in the stack space, meaning the thread will eventually run out of stack space and the application will crash.

A recommended fix for this issue is to ensure that the maximum number of remote players in the session (5) is not less than the amount of steam ID entries included in the packet.

10 - “Join details”

Below is how I understand that this packet data is structured:

```
struct SPacket10 {
	uint32_t iMapID;
	float fSpawnPosX;
	float fSpawnPosY;
	float fSpawnPosZ;
	float fUnk1;
	float fSpawnPosR;
	uint16_t wNominatedWhoId;
	uint16_t wUnk3;
	uint32_t iHostPlayId;
	uint8_t bUnk4;
};
```

This packet is sent to connecting players by the host, and gives important information such as which map ID to spawn in to, the spawn position, as well as the player’s network “WhoID” which is used in other packets to get PlayerCtrl pointers for the purpose of, for example, applying SpEffects to the correct player.

Signalling NaN/Inf

As the spawn position is sent to the other player, it is possible for the host to edit the spawn point on their end. This means that the host can essentially choose where to spawn the player when they join their world.

If the host sends NaN/Inf in this entry, the joining player will be met with a black screen which will not resolve until the game has been fully reset. Even upon returning to their own world, the screen will remain black.

I recommend that spawn positions have some form of sanity check to make sure that they are both on solid ground, and also in valid coordinate parameters. There are some cases where strict client-sided verification is not appropriate (e.g. Looking Glass Knight invasions, where the connecting player needs to know where to spawn from) so this may need to be adapted to accommodate those cases.

12 - “Host world data”

This is an incredibly large packet, and as such it has a large struct with several sub-structs:

```
struct SMapObjCtrl {
	char pUnk1[12];
};

struct SNetNPCPhantom {
	uint32_t iEntityID;
	uint32_t iSummoningState;
	float fTimer;
	uint16_t wNPCSummonSlot;
};

struct SNetNPCPhantomManager {
	uint32_t iCount;
	SNetNPCPhantom sNPCPhantom[5];
	uint32_t iuk;
};

struct SMapObjState {
	char pMapObjState[8];
};

struct SItemDrop {
	uint32_t iUnk;
	uint32_t iItemID;
	float fDurability;
	uint16_t wQuantity;
	uint8_t bUpgrade;
	uint8_t bInfusion;
};

struct SItemWorldInfo {
	SItemDrop sItems[8];
	uint8_t bAmount;
	uint8_t bIsUnknownItemType;
	uint8_t bPrismVfxColour;
	uint8_t bIsGroundItem;
	float fX;
	float fZ;
	float fY;
	float fR;
	uint32_t iMapID;
};
```

```
struct SPacket12 {
	SMapObjCtrl sMapObjCtrl[1024];
	uint32_t iMapObjectCtrlCount;
	SMapObjState sMapObjState[32];
	uint32_t iMapObjStateCount;
	char pEventFlagBuffer[1250];
	char pEventFlagActionCtrl[1326];
	char pEventValueBuffer[1280];
	char pEnemyDeadCtrl[2832];
	char pUnkArray4[8];
	SPlayerDataStruct pPlayerArray[5];
	uint8_t bPlayerCount;
	uint8_t bJunkBytes1[3];
	SNetNPCPhantomManager sNetNPCPhantomManager;
	char pBonfire[128];
	uint8_t bJunkBytes2[4];
	uint64_t qItemBagIDs[32];
	uint64_t qPrismStoneIDs[32];
	SItemWorldInfo sItemBags[32];
	SItemWorldInfo sPrismStones[32];
	uint8_t bItemBagCount;
	uint8_t bPrismStoneCount;
	uint8_t bJunkBytes3[2];
	uint8_t bJunkBytes4[12];
	uint32_t iPlayersInWorld;
	uint32_t iInitialEffectOnPlayerCount;
	uint32_t iUnkVectorSize1;
	uint32_t iUnkVectorSize2;
	uint32_t iActiveBulletCount;
};
```

This packet is sent by the host to connecting clients, and includes everything the client needs to know about the current world state. This ranges from other network players currently in the world, enemy locations / death states, object states, event states, and much more.

There is quite a lot here. Overall this packet has serious problems all relating to a lack of bounds checking. All but 1 of these issues can be resolved by simply making sure the values make sense in the context of the packet.

Issues present in 'SPlayerDataStruct' will be detailed in Packet 13 - "Remote Player Data".

Packet loss

if the packet is never received, the connecting client will hang on the loading screen indefinitely. This cannot be mitigated as there is no method for requesting lost packets again. This is further exacerbated by the packet's size, and due to the fact it's sent **unreliably**.

The solution to this would be to either send it reliably, time out the connection if it is not received, or implement a re-request method for this packet in particular.

Maximum players

The offset ‘iPlayersInWorld’ and ‘bMaxPlayersInWorld’ should always match, however the packet can be modified so that they don’t. In addition to this, ‘iPlayersInWorld’ can be greater than 6, for example if a malicious player set it to MAX_UINT32 then it will attempt to assign 4,294,967,295 players into the world which results in a game crash via stack overflow.

This should be addressed by ensuring that ‘iPlayersInWorld’ and ‘bMaxPlayersInWorld’ are equal, and that the packet is rejected and the connection cancelled if there are more than 5 other network players declared in the session.

MapItemMan objects

Both the offsets ‘bItemBagCount’ and ‘bPrismStoneCount’ relate to the amount of MapItem objects and prism stone objects in the host world. Despite there only being 32 of these that can fit in the packet, these values are not checked and can be set to anything (e.g. 255). This causes stack overflow and crashes the application.

This should be fixed by ensuring that these two values are not greater than 32.

Unchecked vectors/array sizes

The values:

- iInitialEffectOnPlayerCount
- iUnkVectorSize1
- iUnkVectorSize2
- iActiveBulletCount

Are unchecked sizes which, if too large, will crash the application. As I have limited knowledge on what these relate to, I cannot suggest a patch. However they should be capped according to what the game is expecting these sizes to be at maximum.

NetNPCPhantomManager

This segment of the packet contains information about any NPC phantoms or invaders in the host world at the time of joining. There are a few issues with this:

SNetNPCPhantom::iCount declares the amount of NetNPCPhantoms in the packet, this should be 5 at maximum, however any value can be declared here and if it’s larger than 5 then it risks an out of bounds read and crash of the application.

SNetNPCPhantom::SNetNPCPhantom::wNPCSummonSlot relates to which “Summoning slot” the NPC phantom is in. This again, should be a maximum of 5, however any value can be declared. High values will crash the application.

SNetNPCPhantom::SNetNPCPhantom::iSummoningState relates to the “State” in which the NPC phantom is in. (e.g. 0 = Inactive | 1 = Summoning.. | 2 = Summoned | 3 = Leaving..). This is used to write fTimer to an array determined by the value in the state variable. This gives a huge scope for out of bounds write, and potential entrypoint for RCE, as a malicious user could overwrite an entry in a function table to redirect code to a place of their choosing.

All of these issues can be resolved by simple bounds checking to ensure that the values do not exceed the maximum possible declared value.

13 - “Remote player data”

Below is how I understand that this packet data is structured:

```
struct SPlayerDataStruct {
	uint8_t bUnkArray0[0x30];
	float fSpawnX;
	float fSpawnZ;
	float fSpawnY;
	float fSpawnQ;
	uint32_t iPhantomType;
	uint32_t iPlayerID;
	uint32_t iUnk21;
	uint32_t iUnk22;
	uint32_t iUnk23;
	uint32_t iUnk24;
	uint32_t iUnk25;
	uint32_t iLeftHandWeapon1;
	uint32_t iRightHandWeapon1;
	uint32_t iLeftHandWeapon2;
	uint32_t iRightHandWeapon2;
	uint32_t iLeftHandWeapon3;
	uint32_t iRightHandWeapon3;
	uint32_t iHead;
	uint32_t iBody;
	uint32_t iHands;
	uint32_t iLegs;
	uint32_t iUnk1;
	uint32_t iUnk2;
	uint32_t iUnk3;
	uint32_t iUnk4;
	uint32_t iUnk5;
	uint32_t iUnk6;
	uint32_t iRing1;
	uint32_t iRing2;
	uint32_t iRing3;
	uint32_t iRing4;
	uint32_t iConsumable1;
	uint32_t iConsumable2;
	uint32_t iConsumable3;
	uint32_t iConsumable4;
	uint32_t iConsumable5;
	uint32_t iConsumable6;
	uint32_t iConsumable7;
	uint32_t iConsumable8;
	uint32_t iConsumable9;
	uint32_t iConsumable10;
```

```
    uint32_t iSpellSlot1;
    uint32_t iSpellSlot2;
    uint32_t iSpellSlot3;
    uint32_t iSpellSlot4;
    uint32_t iSpellSlot5;
    uint32_t iSpellSlot6;
    uint32_t iSpellSlot7;
    uint32_t iSpellSlot8;
    uint32_t iSpellSlot9;
    uint32_t iSpellSlot10;
    uint32_t iUnkSlot1;
    uint32_t iUnkSlot2;
    uint32_t iUnkSlot3;
    uint32_t iUnkSlot4;
    uint32_t iGestureSlot1;
    uint32_t iGestureSlot2;
    uint32_t iGestureSlot3;
    uint32_t iGestureSlot4;
    uint32_t iGestureSlot5;
    uint32_t iGestureSlot6;
    uint32_t iGestureSlot7;
    uint32_t iGestureSlot8;
    uint8_t bReinforceLevelLWeapon1;
    uint8_t bReinforceLevelRWeapon1;
    uint8_t bReinforceLevelLWeapon2;
    uint8_t bReinforceLevelRWeapon2;
    uint8_t bReinforceLevelLWeapon3;
    uint8_t bReinforceLevelRWeapon3;
    uint8_t bReinforceLevelHead;
    uint8_t bReinforceLevelBody;
    uint8_t bReinforceLevelHands;
    uint8_t bReinforceLevelLegs;
    uint8_t bCustomAttributeLWeapon1;
    uint8_t bCustomAttributeRWeapon1;
    uint8_t bCustomAttributeLWeapon2;
    uint8_t bCustomAttributeRWeapon2;
    uint8_t bCustomAttributeLWeapon3;
    uint8_t bCustomAttributeRWeapon3;
    uint32_t iSlotInfo_Unk1;
    uint32_t iSlotInfo_ActiveEquipLeft;
    uint32_t iSlotInfo_ActiveEquipRight;
    uint32_t iSlotInfo_Unk2;
    uint32_t iSlotInfo_Unk3;
    uint32_t iSlotInfo_Unk4;
    float fCnd_LeftHandWeapon1;
    float fCnd_RightHandWeapon1;
```

```
    float fCnd_LeftHandWeapon2;
    float fCnd_RightHandWeapon2;
    float fCnd_LeftHandWeapon3;
    float fCnd_RightHandWeapon3;
    float fCnd_Head;
    float fCnd_Body;
    float fCnd_Hands;
    float fCnd_Legs;
    float fCnd_Ring1;
    float fCnd_Ring2;
    float fCnd_Ring3;
    float fCnd_Ring4;
    uint8_t bUnkArray2[160];
    uint16_t wUnk1;
    uint16_t wPlayerWho;
    uint8_t bHollowLv;
    uint8_t bCovenant;
    uint8_t bRank_None;
    uint8_t bRank_HeirsOfTheSun;
    uint8_t bRank_BlueSentinels;
    uint8_t bRank_BrotherhoodOfBlood;
    uint8_t bRank_WayOfBlue;
    uint8_t bRank_RatKing;
    uint8_t bRank_BellKeeper;
    uint8_t bRank_DragonRemnants;
    uint8_t bRank_CompanyOfChampions;
    uint8_t bRank_PilgrimsOfDark;
    uint16_t wProgress_None;
    uint16_t wProgress_HeirsOfTheSun;
    uint16_t wProgress_BlueSentinels;
    uint16_t wProgress_BrotherhoodOfBlood;
    uint16_t wProgress_WayOfBlue;
    uint16_t wProgress_RatKing;
    uint16_t wProgress_BellKeeper;
    uint16_t wProgress_DragonRemnants;
    uint16_t wProgress_CompanyOfChampions;
    uint16_t wProgress_PilgrimsOfDark;
    uint8_t bVigour;
    uint8_t bEndurance;
    uint8_t bVitality;
    uint8_t bAttunement;
    uint8_t bStrength;
    uint8_t bDexterity;
    uint8_t bIntelligence;
    uint8_t bFaith;
    uint8_t bAdaptability;
```

```
	uint8_t bStatUnk1;
	uint8_t bStatUnk2;
	char cSteamID[16];
	uint8_t bSteamIdZeroExtend;
	uint64_t qUnk26;
	uint32_t iUnk27;
	float fUnk28;
	float fUnk29;
	float fUnk30;
	uint32_t iUnkSpawnByte;
	uint32_t iUnkSpawnId;
	uint32_t iUnk41;
	uint32_t iUnk42;
	uint32_t iUnk43;
	wchar_t wIGN[16];
	uint8_t pUnkArray5[36];
};

struct SPacket13 {
	SPlayerDataStruct sDataStruct;
	uint32_t iTotalInitialEffectCount;
};
```

The purpose of this packet is to send information about a new player joining the session. When a player does join a session as a guest, the host will send the details about every current member in packet 12. The new joining guest will send a packet 13 to the session to allow their game clients to construct the new member.

Invalid covenant IDs

The game uses the ‘bNewCovenant’ to access an array in order to set the ‘IsDiscovered’ bool for the respective covenant. There is no bounds checking here, and any value greater than 9 will result in an out of bounds write and potential crash of the application, depending how far the write is and whether it writes over any important data structures.
To fix this issue, strict bounds checking should be run before accessing this (or any) array. This is identical to the exploit in packet 91.

Invalid active weapon slots

The variables ‘ iSlotInfo_ActiveEquipLeft’ and ‘iSlotInfo_ActiveEquipRight‘ should only ever be 0, 1, or 2 (as there are 3 weapon slots per hand available). This value is used to access an array, so setting it to a value such as MAX_UINT32 will cause an out of bounds read and crash of the application.

I recommend that you assert that this value is less than 3.

Invalid ReinforceLv

Reinforce levels for most weapons range from 0 - 10 in Dark Souls II. Any value greater than this results in a game crash whenever the player swings the weapon. It is possible for a malicious player to send a game packet saying that they have equipped, for example, a scimitar+12, then when they swing the weapon all other players in the session will crash.

This applies to the fields from ‘bReinforceLevelLWeapon1’ to ‘bReinforceLevelLegs’ (10in total).

To fix this, I recommend that you ensure that this value is less than or equal to 10 which is the maximum reinforce level in Dark Souls II.

Invalid infusions (“CustomAttributes”)

Infusions (e.g bleed, poison, raw, mundane, etc) range from a value of 0 - 9. Any value greater than this may cause the game engine to panic with the debug string:

“不正なカスタム属性[%u]が設定されています。
バグなのでプログラムを修正してください”
“N:\FRPG2_64\source\FRPG2\Title\Source\Game\Item2\ItemInventory2BasicTypes.cpp”

To fix this, I recommend checking these values as they come in, and declining values greater than 9. This applies to the fields from bCustomAttributeLWeapon1 to ‘bCustomAttributeRWeapon3’ (6 in total)

Format string

The string in ‘wIGN’ can be formatted. If this formatting is invalid, the game engine will panic and the game will crash. For example, valid formatting may be:

#c[000000]PlayerName#c

Invalid formatting which would crash may look incomplete such as:

#c[000000PlayerName

To fix this issue, format characters like “[“, and “#” should be filtered out and disallowed from player names.

WhoID spoofing

The ID specified in ‘wPlayerWho’ can be modified by a malicious user which will cause numerous in-game bugs (such as irregular damage, SpEffects, and grab attacks).

16 - “Notify leave”

Below is how I understand that this packet data is structured:

```
struct SPacket16 {
	uint8_t bPhantomType;
	char pSteamID64[16];
	wchar_t wCharName[32];
};
```

The purpose of this packet is to inform other players of player disconnections, and is also used when the host uses the black crystal.

Sending home non-friendly players

When a game client receives this packet, it will check to see if the string in ‘pSteamID64’ matches the local player’s SteamID. If it does, it will disconnect you from the host world. This can be manipulated by sending a target player (e.g. a black ghost) this packet directly and send them home from the world, regardless of phantom type.

This can be fixed by implementing two fixes:
1. Making sure the local player’s phantom type is allowed to be expelled by the host before disconnecting from the session.
2. Making sure that this packet is coming from the host.

Format string

The string in ‘wCharName’ can be formatted. If this formatting is invalid, the game engine will panic and the game will crash. For example, valid formatting may be:

#c[000000]PlayerName#c

Invalid formatting which would crash may look incomplete such as:

#c[000000PlayerName

To fix this issue, format characters like “[“, and “#” should be filtered out and disallowed from player names.

17 - “Leave message”

Below is how I understand that this packet data is structured:

```
struct SPacket17 {
	uint8_t bUnk1;
	uint8_t bUnk2;
	wchar_t wLeaveName[32];
};
```

This packet is sent by players who are disconnecting from a session and makes the “Phantom (PlayerName) has returned to their world” message appear.

Format string

Problems present in “Packet 16 - “Notify Leave” are also present here.

NetEnemyManager

This ranges from packet ID 20 - 24. I will merge all 4 packets into one instance since they share the same issues:

```
struct SEnemyLocationData {
	float fX;
	float fZ;
	float fY;
	uint16_t wUnk1;
	uint16_t wUnk2;
	uint32_t iID;
};

struct SPacket20 {
	SEnemyLocationData sEntry[8];
};

struct SPacket21 {
	uint16_t wUnk1;
	uint8_t bID;
	uint8_t bUnk2;
};

struct SPacket22 {
	uint16_t wUnk1;
	uint8_t bID;
	uint8_t bPadding;
};

struct SPacket23 {
	uint32_t iUnk8;
	uint16_t wUnk3;
	uint8_t bUnk4;
	uint8_t bUnk5;
	uint32_t iUnk9;
	uint16_t wUnk6;
	uint16_t wUnk7;
	uint8_t bUnk[2];
	uint16_t wUnk8;
	uint8_t bID;
	uint8_t bUnk2;
};

struct SPacket24 {
	uint8_t bID;
	uint8_t bFlag;
```

```
};
```

The information I have for these packets is limited, so I apologise. The major issue with all of these is the ‘bID’ field. This is used to access an array which can lead to an out-of-bounds write.

I recommend that you ensure that the ‘bID’ declared does not exceed the enemy count in the area that the player is in.

BulletEmitterPacketReceiver

27 - “Turret”

Below is how I understand that this packet data is structured:

```
struct SPacket27 {
	uint32_t iWhoIsSource;
	uint32_t iWhoIsTarget;
	uint32_t iBulletParamEntry;
	uint32_t iDamageParamEntry;
	float fUnk1;
	float fUnk2;
	uint8_t bUnk3;
	uint8_t bIsUseAbsoluteEntityId;
	uint8_t bUnk4;
	uint8_t bUnk5;
	float fUnk6;
	float fUnk7;
	float fUnk8;
	float fUnk9;
	float fUnk10;
	float fUnk11;
	float fUnk12;
	float fUnk13;
	float fUnk14;
	float fUnk15;
	float fUnk16;
	float fUnk17;
	float fUnk18;
	float fUnk19;
	float fUnk20;
	float fUnk21;
	float fUnk22;
	float fUnk23;
	float fUnk24;
	float fUnk25;
	float fUnk26;
	float fUnk27;
	float fUnk28;
	float fUnk29;
	float fUnk30;
	float fUnk31;
	float fUnk32;
	float fUnk33;
	float fUnk34;
	float fUnk35;
```

```
        float fUnk36;
        float fUnk37;
        float fUnk38;
        float fUnk39;
        float fUnk40;
        float fUnk41;
        float fUnk42;
        float fUnk43;
        uint32_t iUnk44;
};
```

Excessive bullets

The most common exploit players use in Dark Souls II to crash others is use of excessive bullets. By sending 100's of bullet packets at once, the game engine will panic due to being 'out of memory.'

To fix this, I recommend a strict bullet cap across the entire game.

WhoID spoofing

The ID specified in 'iWhoIsSource' determines which player character to spawn the emitted bullet from. Players can change this to mimic other players, resulting in any bullet being emitted from any player at any time.

I recommend this be fixed by disallowing the local player's whoid to be referenced in this packet when being received.

53 - "Turret (enemy)"

This packet is structurally identical, and has the same issues as packet 27 (see above).

ChrDamagePacketReceiver

28 - “PvP damage”

Below is how I understand that this packet data is structured:

```
struct SDamagePacket {
	uint32_t iWhoIsHit;
	uint32_t iWhoIsHitter;
	uint32_t iPlayerDamageParam;
	uint32_t iUnk1;
	float fPhysicalDamage;
	float fMagicDamage;
	float fLightningDamage;
	float fFireDamage;
	float fDarkDamage;
	float fUnkStatusEffectDamage1;
	float fPoisonDamage;
	float fBleedDamage;
	float fUnkStatusEffectDamage2;
	float fCurseDamage;
	float fUnkStatusEffectDamage3;
	float fToxicDamage;
	float fPetrifyDamage;
	float fPostureDamage;
	float fPhysicalDamageMultiplier1;
	float fPhysicalDamageMultiplier2;
	float fUnk2;
	float fUnk3;
	float fDurabilityDamageMultiplier;
	float fUnk5;
	float fUnk6;
	float fUnk7;
	float fUnk8;
	float fUnk9;
	float fUnkStatusEffectMulti1;
	float fPoisonMulti;
	float fBleedMulti;
	float fUnkStatusEffectMulti2;
	float fCurseMulti;
	float fUnkStatusEffectMulti3;
	float fToxicMulti;
	float fPetrifyMulti;
	float fHitSoundSrcX;
	float fHitSoundSrcZ;
	float fHitSoundSrcY;
	float fStaggerDirX;
```

```
    float fStaggerDirZ;
    float fStaggerDirY;
    uint16_t wUnk300;
    uint16_t wUnk301;
    uint8_t bUnk200;
    uint8_t bUnk201;
    uint16_t wUnk202;
    uint8_t bUnk100;
    uint8_t bUnk101;
    uint8_t bUnk102;
    uint8_t bUnk103;
    uint16_t wUnk29;
    uint8_t bDamageMotionWeightParamEntry;
    uint8_t bUnk31;
    uint8_t bUnk32;
    uint8_t bUnk33;
};
```

Packet 28 is a damage packet that is used to communicate PvP damage between clients.

Unobtainable status effects

Malicious players use common cheats, such as setting ‘fPetrifyDamage’ to an infinite number which causes instant build-up. Normally, players cannot apply these effects through normal means so they should be disallowed.

I suggest that this packet prevents communicating status effects such as petrify and curse which cannot be performed by players in the first place.

WhoID spoofing

The ID specified in ‘iWhoIsHit’ determines which player ID takes the hit in the damage packet. If someone changes this ID or mimics someone else’s ID, then it causes abnormal game behaviour where other players take damage instead of them.

I recommend this be fixed by only sending the damage packet to the player who was hit, instead of sending it to the whole session and removing the ‘iWhoIsHit’ variable completely, similar to the networking found in Dark Souls 1 and Dark Souls 3.

29 - “PvE damage”

Below is how I understand that this packet data is structured:

```
struct SPacket29 {
	SDamagePacket sDamage;
};
```

Packet 29 is structurally identical to packet 28. It has the same function but synchronises PvE damage as opposed to PvP damage. See above

30 - “Unknown damage (PvP)”

Below is how I understand that this packet data is structured:

```
struct SPacket30 {
	SDamagePacket sDamage;
};
```

Packet 30 is structurally identical to packet 28. See above.

31 - “Unknown damage (PvE)”

Below is how I understand that this packet data is structured:

```
struct SPacket31 {
	uint32_t iPacking;
	uint16_t wWhoIs;
	uint16_t wPadding;
	uint8_t bPackedIntegerCount;
	uint32_t pValues[4];
};
```

I’m unsure about this packet’s function. I have never seen it used in-game. Despite this, it can still be received by other players.

Fixed vector crash

The value ‘bPackedIntegerCount’ relates to the amount of entries declared in ‘pValues’. Despite the maximum amount being less than 4, a malicious player can set this value to a value greater than 4. This causes the receiving player’s game engine to panic with the debug string:

“追加できる最大数を越えています(最大:%d 現在:%d)”
“N:\FRPG2_64\source\FRPG2\Title\Source\Network/Util/NvdFixedVector.h”

NetP2pPacketEventFlag

32 - “Event flag”

```
struct SPacket32{
	uint32_t iEventID;
	uint8_t bFlagState;
};
```

This packet is used to synchronise some event flags between clients.

Invalid set/unset flags

Malicious players can join the host world and set or unset **any** flag on the host's game. In theory, players could completely reset the game state, or set every single event flag on another player's game which could ruin or corrupt their game file.
Note that this only makes a difference when it is on the host game. Remote players are not affected as, when they return to their own world, the world state is restored.

I suggest that either this packet be removed and an alternative created, or the game makes sure the event flag is able to be set by a remote player.

Blue Acolyte resolves this issue by ignoring this packet on the host game, and accepting it if it is a remote player game

BulletPacketReceiver

40 - “Player bullet”

Below is how I understand that this packet data is structured:

I do not have a complete structure for this packet.

Excessive bullets

The most common exploit players use in Dark Souls II to crash others is use of excessive bullets. By sending 100’s of bullet packets at once, the game engine will panic due to being ‘out of memory.’

To fix this, I recommend a strict bullet cap across the entire game.

ChrLockOnPacketReceiver

46 - ”PvP lock on”

Below is how I understand that this packet data is structured:

```
struct SPacket46 {
	uint32_t iWhoIsLockedOn;
	uint32_t iWhoIsLocked;
};
```

This packet is sent whenever a player uses the target functionality on another player.

WhoID spoofing

The player can change ‘iWhoIsLockedOn’ and ‘iWhoIsLocked’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet.

I recommend this be fixed by ensuring the ‘iWhoIsLockedOn’ entry player ID does not match the local player.

47 - ”PvE lock on”

Below is how I understand that this packet data is structured:

```
struct SPacket47 {
	uint32_t iWhoIsLockedOn;
	uint32_t iWhoIsLocked;
};
```

This packet is sent whenever a player uses the target functionality on an NPC.

WhoID spoofing

The player can change ‘iWhoIsLockedOn’ and ‘iWhoIsLocked’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns/EnemyIns pointer for the target of the function of the packet.

I recommend this be fixed by ensuring the ‘iWhoIsLockedOn’ entry player ID does not match the local player.

NetP2pPacketSpEffect

49 - “SpEffect sync”

Below is how I understand that this packet data is structured:

```
struct SPacket49 {
	uint32_t iUnk1;
	uint32_t iWhoIs;
	uint32_t iSpEffectID;
	uint32_t iSpEffectType;
	float fSpEffectDuration;
};
```

This packet is used to communicate SpEffects between clients. For example if player A uses a lifegem, it will send a packet 49 to the rest of the session with Player A’s ‘WhoId’, and details about the SpEffect that they have applied to themselves.

WhoID spoofing

The player can change ‘iWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns/EnemyIns pointer for the target of the function of the packet.

I recommend this be fixed by discarding the ‘iWhoIs’ argument, and instead always applying the received SpEffect info to the PlayerIns entry associated with the NetworkPlayer entry who sent the packet.

Stacking effects

While applying effects to the player, there is an entry that allows you to apply multiple effects, multiple times. In this case, for example Player A could use a cheat that applies the “Gower's Ring of Protection” effect to themselves 100,000 times. In this case, other players in the session will crash due to insufficient memory, as Player A’s game client will send 100,000 SpEffect packets at once and overload the receiving game clients.

This can be fixed by limiting the amount of SpEffects a player can have at once.

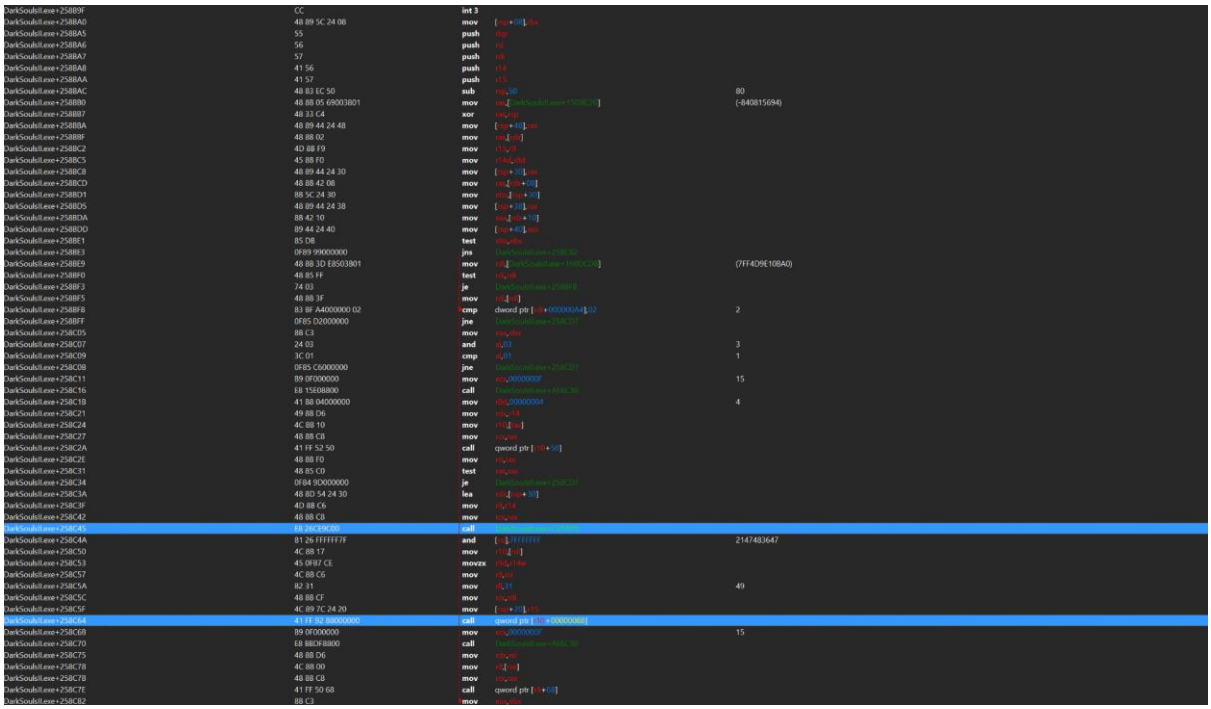
Out-of-bounds stack read and send

func_0x258BA0 is the function which parses the received SpEffect packet. With certain arguments, this function will transmit the packet data back to the client who sent it.

The issue is that the size portion is unchecked. This means that a malicious player could send a 1000 byte packet to a player, and their game client would send 1000 bytes of the stack contents back to the player who sent the packet originally.

If this value is too high that it exceeds the stack size, it will crash. Otherwise it will send important stack contents back to the malicious player (e.g. static memory addresses) who could then use it to defeat the purpose of ASLR (address-space layout randomisation). This makes the possibility of a **reliable RCE attack** substantially more likely, as the malicious player can now reliably deduce where the executable is in memory.

Below is the function at fault. The size argument for the packet is in register r14. The function at [r10+88] is SendP2PPacketToPlayer. The highlighted function (DarkSoulsII.exe+C25A70) is memcpy.



ChrGrabPacketReceiver

51 - “Grab”

Below is how I understand that this packet data is structured:

```
struct SThrowDamageStruct {
	uint32_t iSystemDamageParamID;
	float fPhysicalDamage;
	float fMagicDamage;
	float fLightningDamage;
	float fFireDamage;
	float fDarkDamage;
	float fUnkStatusEffectDamage1;
	float fPoisonDamage;
	float fBleedDamage;
	float fUnkStatusEffectDamage2;
	float fCurseDamage;
	float fUnkStatusEffectDamage3;
	float fToxicDamage;
	float fPetrifyDamage;
	float fUnk1;
	float fUnk2;
	float fUnk3;
};

struct SPacket51 {
	int iThrowerWhoIs;
	int iThroweeWhoIs;
	uint32_t iGrabParamID;
	uint32_t iUnkID;
	uint32_t iUnk1;
	uint32_t iUnk2;
	uint16_t wThrowerAnim;
	uint16_t wThroweeAnim;
	uint16_t wEncodedThrowerX;
	uint16_t wEncodedThrowerZ;
	uint16_t wEncodedThrowerY;
	uint16_t wEncodedThrowerR;
	uint16_t wEncodedThroweeX;
	uint16_t wEncodedThroweeZ;
	uint16_t wEncodedThroweeY;
	uint16_t wEncodedThroweeR;
	SThrowDamageStruct sDamagestructs[3];
};
```

This packet is used to communicate throw / grab attacks (e.g. backstabs, ripostes, boss grabs) between clients.

WhoID spoofing

The player can change both the ‘iThrowerWhoIs’ and ’iThroweeWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns/EnemyIns pointer for the target of the function of the packet.

The result of this is, for example, if we have 3 players in a session: Player A, Player B, and Player C. Player C could send a packet 51 to the session with Player A’s who ID in the “Thrower” slot, and Player B’s who ID in the “Throwee” slot. This would cause Player A to perform a grab attack on Player B with absolutely no interaction from those players whatsoever.

This can also be used to essentially lock players in infinite grab loops. For example by making Player A both the thrower and throwee, causing them to grab themselves.

Unobtainable status effects

As this packet also contains information for damaging other players, it shares the same pitfalls as the PvP damage packet (packet 28) which allows players to apply otherwise unobtainable status effects for the player (such as petrify and curse) to other players via damage.

Malicious players use common cheats, such as setting ‘fPetrifyDamage’ to an infinite number which causes instant build-up. Normally, players cannot apply these effects through normal means so they should be disallowed.

I suggest that this packet prevents communicating status effects such as petrify and curse which cannot be performed by players in the first place.

ChrDeadPacketReceiver

54 - “ChrDeadActionCtrl (PC)”

Below is how I understand that this packet data is structured:

```
enum EDeathType : uint8_t {
	eDeathType_StandDeath_01 = 10,
	eDeathType_FallDeath_01 = 20,
	eDeathType_FireDeath_01 = 30,
	eDeathType_StandDeath_02 = 40,
	eDeathType_PetrifyDeath_01 = 50,
	eDeathType_Pancake_01 = 60,
	eDeathType_PetrifyDeath_02 = 70,
	eDeathType_Ladder_01 = 80,
	eDeathType_FallDeath_03 = 90,
	eDeathType_FallDeath_04 = 100,
	eDeathType_PetrifyDeath_03 = 110,
	eDeathType_FallDeath_05 = 120,
	//130
	//140
	eDeathType_FallDeath_06 = 150,
	eDeathType_PoisonDeath_01 = 160,
	eDeathType_PoisonDeath_02 = 170,
	eDeathType_Pancake_02 = 180,
};

struct SPacket55 {
	uint32_t iWhoIsDead;
	uint32_t iWhoIsKiller;
	uint32_t iUnk2;
	uint32_t iUnk3;
	uint32_t iUnk4;
	uint16_t wUnk5;
	EDeathType eDeathType;
	uint8_t bUnk7;
};
```

Packet description goes here?

WhoID spoofing

A malicious player can change ‘iWhoIsDead’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can force a death state on any other player. This can sometimes lead to them getting trapped in the session infinitely unable to die or disconnect.

I recommend this be fixed by ensuring the received player ID does not match the local player.

55 - “ChrDeadActionCtrl (NPC)”

Below is how I understand that this packet data is structured:

```
enum EDeathType : uint8_t {
	eDeathType_StandDeath_01 = 10,
	eDeathType_FallDeath_01 = 20,
	eDeathType_FireDeath_01 = 30,
	eDeathType_StandDeath_02 = 40,
	eDeathType_PetrifyDeath_01 = 50,
	eDeathType_Pancake_01 = 60,
	eDeathType_PetrifyDeath_02 = 70,
	eDeathType_Ladder_01 = 80,
	eDeathType_FallDeath_03 = 90,
	eDeathType_FallDeath_04 = 100,
	eDeathType_PetrifyDeath_03 = 110,
	eDeathType_FallDeath_05 = 120,
	//130
	//140
	eDeathType_FallDeath_06 = 150,
	eDeathType_PoisonDeath_01 = 160,
	eDeathType_PoisonDeath_02 = 170,
	eDeathType_Pancake_02 = 180,
};

struct SPacket55 {
	uint32_t iWhoIsDead;
	uint32_t iWhoIsKiller;
	uint32_t iUnk2;
	uint32_t iUnk3;
	uint32_t iUnk4;
	uint16_t wUnk5;
	EDeathType eDeathType;
	uint8_t bUnk7;
};
```

WhoID spoofing

Similar to the above (packet 54), the player can change ‘iWhoIsDead’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can force a death state on any other player. This can sometimes lead to them getting trapped in the session infinitely unable to die or disconnect.

I recommend this be fixed by ensuring the received player ID does not match the local player. In addition to this, you could be even stricter and ensure that the ID is an NPC, and not a player at all.

ChrEquipPacketReceiver

Parsing function: sub_0x15F4E0

58 - “Weapon stance”

Below is how I understand that this packet data is structured:

```
struct SPacket58 {
	uint32_t iPlayerWhoIs;
	uint8_t bStance;
};
```

This packet is used to network stances (e.g one hand, two hand, power-stancing, etc.) done by other players.

WhoID spoofing

The player can change ‘iPlayerWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can change the stance of the target player.

This is also related to a common game bug. If a host player walks over a summon sign, they will send a “Weapon stance” packet to the session with a ‘iPlayerWhoIs’ of 0x3FFF. This id will **always** return a pointer to the local player PlayerIns pointer and forcibly change the weapon stance of every player in the session to default 1-handed. This is because the host’s game client attempts to send a “Weapon stance” packet to the session about the PlayerPreviewChar when walking over the summon sign.

Invalid stances

Valid stance values are 0 - 6. Any higher value will cause an out of bounds write. This needs to be asserted as < 7.

59 - “Active weapon change”

Below is how I understand that this packet data is structured:

```
struct SPacket59 {
	uint32_t iPlayerWhoIs;
	uint8_t bHand;
	uint8_t bActiveWeaponSlot;
};
```

This packet is used to network player weapon slot or “Hand” changes (e.g. a player switching from right weapon slot 1 to right weapon slot 2).

WhoID spoofing

The player can change ‘iPlayerWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can change the active weapon slot of the target player.

False banning / Out of bounds write / Save file corruption

The variable ‘bHand’ is used to access an array in which to write the variable ‘bActiveWeaponSlot’. ‘bHand’ should only ever be 0, or 1 (as the player has a right and a left hand), however values like 6 will write to the player’s pointer for gender, which will cause them to be flagged and banned by the servers for tampering with save data.

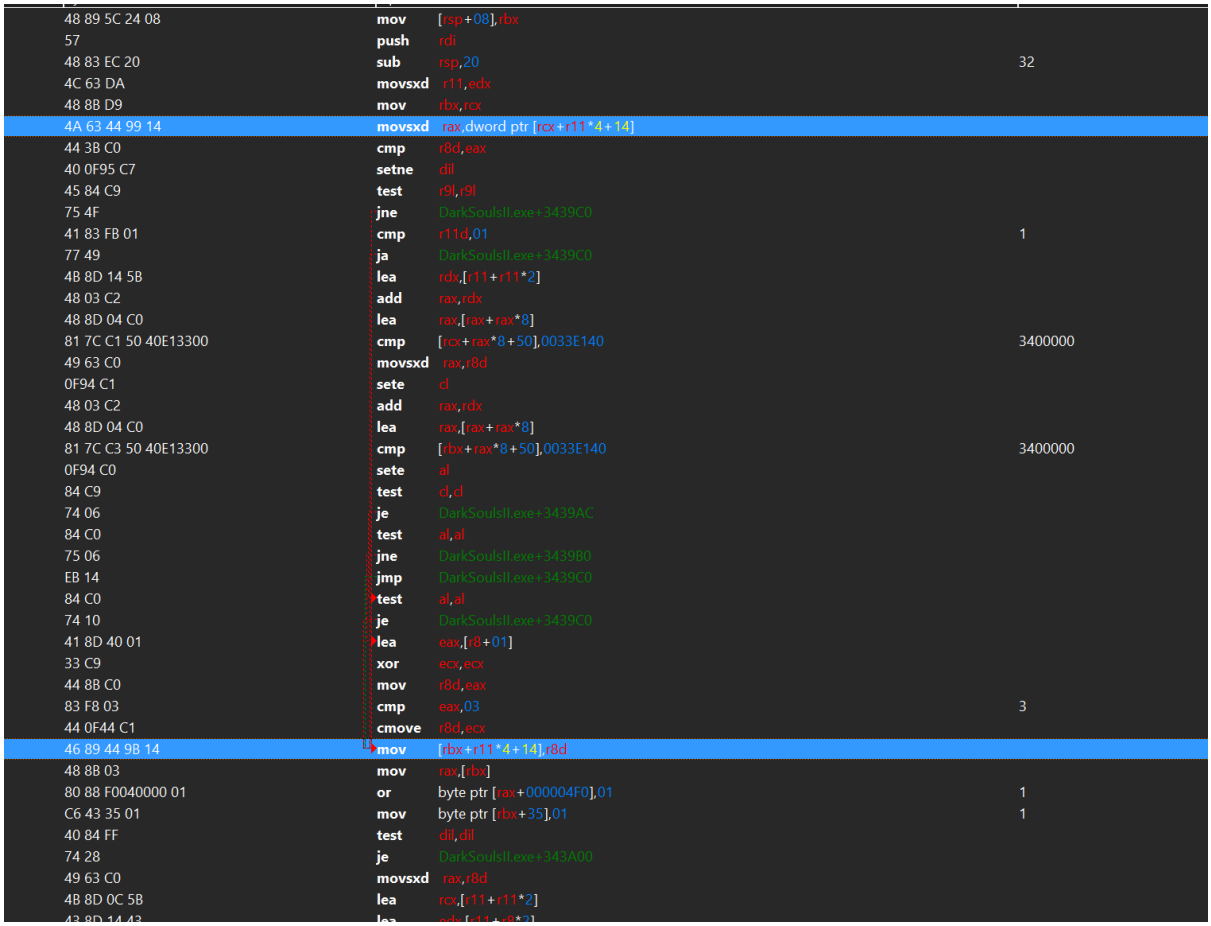
I recommend that you assert this value is less than 2, in addition to the patches mentioned above so that the local player’s data cannot be modified by remote players in the first place.

Save file corruption (2)

The variable ‘bActiveWeaponSlot’ should only ever be 0, 1, or 2 (as there are 3 weapon slots per hand available). However, if changed to a value like 4, or 6, it causes the target player to be unable to break out of the invalid weapon slot. Whenever they open their inventory menu, the game will crash. This value is saved to their profile, so they are unable to undo the change and the save file is corrupted.

I recommend that you assert that this value is less than 3.

Below is a screenshot of the function where the arrays are accessed and written to (sub_0x343950). In this case ‘r11’ is ‘bHand’ and ‘r8d’ is ‘bActiveWeaponSlot’. Note the lack of boundary checking.



```
48 89 5C 24 08          mov     [rsp+08],rbx
57                      push    rdi
48 83 EC 20             sub     rsp,20                          32
4C 63 DA                movsxd  r11,edx
48 8B D9                mov     rbx,rcx
4A 63 44 99 14          movsxd  rax,dword ptr [rcx+r11*4+14]
44 3B C0                cmp     r8d,eax
40 0F95 C7              setne   dil
45 84 C9                test    r9l,r9l
75 4F                   jne     DarkSoulsII.exe+3439C0
41 83 FB 01             cmp     r11d,01                         1
77 49                   ja      DarkSoulsII.exe+3439C0
4B 8D 14 5B             lea     rdx,[r11+r11*2]
48 03 C2                add     rax,rdx
48 8D 04 C0             lea     rax,[rax+rax*8]
81 7C C1 50 40E13300    cmp     [rcx+rax*8+50],0033E140         3400000
49 63 C0                movsxd  rax,r8d
0F94 C1                 sete    cl
48 03 C2                add     rax,rdx
48 8D 04 C0             lea     rax,[rax+rax*8]
81 7C C3 50 40E13300    cmp     [rbx+rax*8+50],0033E140         3400000
0F94 C0                 sete    al
84 C9                   test    cl,cl
74 06                   je      DarkSoulsII.exe+3439AC
84 C0                   test    al,al
75 06                   jne     DarkSoulsII.exe+3439B0
EB 14                   jmp     DarkSoulsII.exe+3439C0
84 C0                   test    al,al
74 10                   je      DarkSoulsII.exe+3439C0
41 8D 40 01             lea     eax,[r8+01]
33 C9                   xor     ecx,ecx
44 8B C0                mov     r8d,eax
83 F8 03                cmp     eax,03                          3
44 0F44 C1              cmove   r8d,ecx
46 89 44 9B 14          mov     [rbx+r11*4+14],r8d
48 8B 03                mov     rax,[rbx]
80 88 F0040000 01       or      byte ptr [rax+000004F0],01      1
C6 43 35 01             mov     byte ptr [rbx+35],01            1
40 84 FF                test    dil,dil
74 28                   je      DarkSoulsII.exe+343A00
49 63 C0                movsxd  rax,r8d
4B 8D 0C 5B             lea     rcx,[r11+r11*2]
```

60 - “Equipment change”

Below is how I understand that this packet data is structured:

```
struct SPacket60 {
	uint32_t iPlayerWhoIs;
	uint32_t iEquipmentID;
	uint8_t bHand;
};
```

I’ve never seen this packet used. I recommend that it is removed as the malicious potential is very large.

WhoID spoofing

The player can change ‘iPlayerWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can change the visuals of the equipment that the target player is holding.

I recommend this be fixed by preventing remote players from changing the equipment properties of the local player, and rejecting packets that attempt to do so.

Out of bounds write

The variable ‘bHand’ is used to access an array. It is passed to this function (sub_0x343AD0) as an argument in ‘edx’. It’s then multiplied by 16 where it is then used extensively throughout the function to read and write from that array offset.

```
mov     [rsp+18],rbx
mov     [rsp+20],rbp
push    rsi
push    rdi
push    r14
sub     rsp,20
movsxd  r14,edx
mov     ebp,r8d
mov     rbx,rcx
lea     rdi,[r14+r14*8]
cmp     [rcx+rdi*8+00000200],r8d
je      DarkSoulsII.exe+343C4E
mov     rax,[DarkSoulsII.exe+160B8D0]
xor     edx,edx
mov     [rsp+40],r12
mov     r12,[rax+18]
mov     [rsp+48],r15
lea     r15,[rbx+rdi*8]
```

61 - “Remote weapon change”

Below is how I understand that this packet data is structured:

```
struct SPacket61 {
	struct SWeaponEquipDetails {
		uint8_t bByte; // EquipmentSlot & IsBroken
		uint8_t bReinforce; // CustomAttribute & ReinforceLv
	};

	uint32_t iWeaponID;
	SWeaponEquipDetails sWeaponAttributes;
};
```

This packet is sent when a remote player changes their weapon, so that it updates on their player for everyone else in the session.

Invalid infusions (“CustomAttributes”)

Infusions (e.g bleed, poison, raw, mundane, etc) range from a value of 0 - 9. Any value greater than this may cause the game engine to panic with the debug string:

“不正なカスタム属性[%u]が設定されています。
バグなのでプログラムを修正してください”
“N:\FRPG2_64\source\FRPG2\Title\Source\Game\Item2\ItemInventory2BasicTypes.cpp”

To fix this, I recommend checking these values as they come in, and declining values greater than 9.

Invalid ReinforceLv

Reinforce levels for most weapons range from 0 - 10 in Dark Souls II. Any value greater than this results in a game crash whenever the player swings the weapon. It is possible for a malicious player to send a game packet saying that they have equipped, for example, a scimitar+12, then when they swing the weapon all other players in the session will crash.

(BUG) Non-updating breakage visuals on weapons

If a player equips a weapon (e.g. a scimitar) which is broken, then equips another scimitar that is not broken, the visuals do not update for other players. This is because the function to refresh the player’s equipment (sub_0x341F30) is never called once the new weapon is equipped, which creates the bugged visuals.

62 - “Remote armour change”

Below is how I understand that this packet data is structured:

```
struct SPacket62 {
	struct SArmourEquipDetails {
		uint8_t bByte; // EquipmentSlot & IsBroken
		uint8_t bReinforce;
	};
	uint32_t iArmourID;
	SArmourEquipDetails sInfo;
};
```

This packet is sent when a remote player changes their armour, so that it updates on their player for everyone else in the session.

Incorrect equipment slots

As the packet defines the slot in which to equip the armour piece, it is possible for some incorrect combinations (e.g. Hexer hood equipped in the chest slot). Some of these combinations cause the game to crash.
To address this issue, I recommend that the equipment slot is either omitted and checked client-sided (via ArmorParam) to ensure that it is valid.

64 - “Equipment status”

Below is how I understand that this packet data is structured:

```
#pragma pack(push,1)
struct SEquipmentStatus {
	uint16_t wOffset; // This is signed, but we don't want it to go
backwards!
	EEquipmentStatusSlot eType;	// 0 = Weapon | 1 = Armour | 2 =
Ring
	uint8_t bAction;  // 1 = Broken | 2 = Repaired
};
#pragma pack (pop)
struct SPacket64 {
	uint32_t iPlayerWhoIs;
	uint8_t bPackedIntegerCount;
	SEquipmentStatus sData[14];
};
```

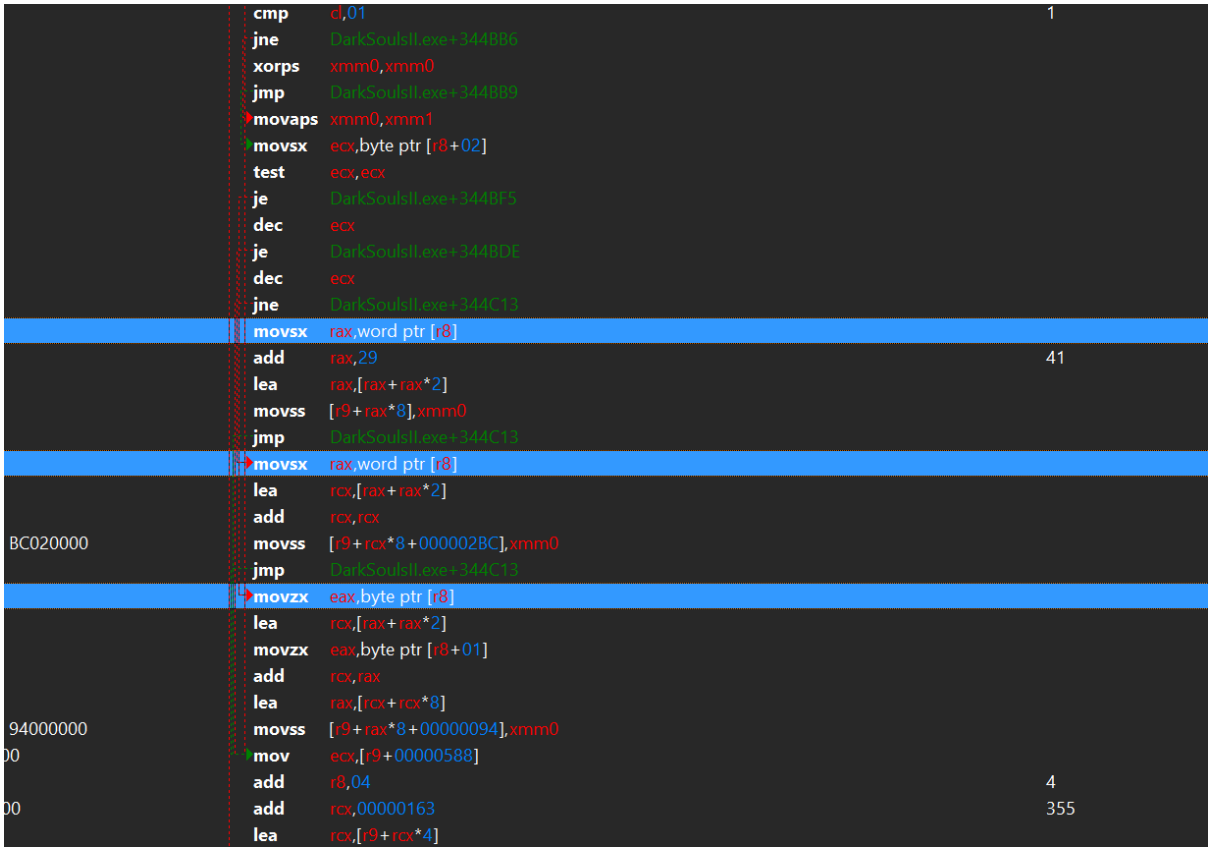
This packet is sent to create equipment breakage vfx, and update visuals on other player’s games after the session has already begun (e.g. a player breaking their weapon mid-fight)

Out of bounds write

The variable ‘wOffset’ is used to access an array of floats which points to the location of the ‘durability’ of the piece of equipment to break. There is no bounds checking on this value, and there is potential for a huge out of bounds write.

To fix this, there simply needs to be boundary checking on these values. For example there are only 4 valid armour slots (0, 1, 2, and 3) so the ‘wOffset’ for armour breakage should be below 4, while for weapons the valid values for wOffset are between 0 and 5.

Here is the function which reads from the fixed vector (sub_0x344B70). Note the highlighted instructions are accessing the value ‘wOffset’ and using that value to insecurely access player equipment data.



Fixed vector crash

The value ‘bPackedIntegerCount’ relates to the amount of entries declared in ‘sData’. Despite the maximum amount being 14 or below, a malicious player can set this value to a value greater than 14. This causes the receiving player’s game engine to panic with the debug string:

“追加できる最大数を越えています(最大:%d 現在:%d)”
“N:\FRPG2_64\source\FRPG2\Title\Source\Network/Util/NvdFixedVector.h”

I recommend that you ignore packets with invalid data like this, instead of calling the panic function. Below is a screenshot of the fixed vector function at fault (sub_0x344280):

```
sub     rsp,38                                  56
movzx   eax,byte ptr [rdx]
xor     r9d,r9d
mov     r11,rdx
lea     r8,[rdx+01]
mov     [rcx],r9d
mov     r10d,eax
test    eax,eax
jle     DarkSoulsII.exe+3442BC
nop     dword ptr [rax+rax+00]
mov     edx,[rcx]
mov     eax,[r8]
add     r8,04                                   4
cmp     edx,0E                                  14
jae     DarkSoulsII.exe+3442C7
inc     r9d
mov     [rcx+rdx*4+04],eax
inc     [rcx]
cmp     r9d,r10d
jl      DarkSoulsII.exe+3442A0
sub     r8,r11
mov     eax,r8d
add     rsp,38                                  56
ret
lea     eax,[rdx+01]
mov     r9d,0000000E                            14
lea     r8,[DarkSoulsII.exe+10BA9F0]            (-17.22)
lea     edx,[r9+1D]
lea     rcx,[DarkSoulsII.exe+10BAA20]           ("N:\FRPG2_
mov     [rsp+20],eax
call    DarkSoulsII.exe+82B750
```

WhoID spoofing

The player can change ‘iPlayerWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can apply the effect of this packet to whoever they like in the session.

To address this issue, ‘iPlayerWhoIs’ should be strictly checked to ensure that it does not result in the PlayerIns pointer for the local player.

Visual effect crashes (memory corruption)

A malicious user can simply send this packet over and over extremely quickly, telling other clients that all 14 equipment slots are breaking over and over. This makes a rather loud sound, and the visuals from the equipment sparks quickly crashes other player’s games. I have attached a YouTube video demonstrating this.

To resolve this, I recommend you check whether the equipment is indeed breakable before allowing it to be broken again. Malicious players can get around this by rapidly repairing and breaking their weapon too, so I also recommend more robust checks and limiting the amount of these packets you can receive from each player per unit time.

In this video, I demonstrate both the visual effect crash, and the WhoID spoofing exploit. The red phantom is the malicious player sending the packets, and the host is not using any cheats:

https://www.youtube.com/watch?v=qOcpy6v2_wY

65 - “Equipment condition”

Below is how I understand that this packet data is structured:

```
enum EVisibleDamageFlags : uint32_t {
	eVisibleDamageFlags_LWeapon1 = 1 << 0,
	eVisibleDamageFlags_RWeapon1 = 1 << 1,
	eVisibleDamageFlags_LWeapon2 = 1 << 2,
	eVisibleDamageFlags_RWeapon2 = 1 << 3,
	eVisibleDamageFlags_LWeapon3 = 1 << 4,
	eVisibleDamageFlags_RWeapon3 = 1 << 5,
	eVisibleDamageFlags_Head = 1 << 6,
	eVisibleDamageFlags_Body = 1 << 7,
	eVisibleDamageFlags_Hand = 1 << 8,
	eVisibleDamageFlags_Legs = 1 << 9,
	eVisibleDamageFlags_Ring1 = 1 << 10,
	eVisibleDamageFlags_Ring2 = 1 << 11,
	eVisibleDamageFlags_Ring3 = 1 << 12,
	eVisibleDamageFlags_Ring4 = 1 << 13,
};

struct SPacket65 {
	struct SVisibleEquipmentDamage {
		uint32_t iPlayerWhoIs;
		EVisibleDamageFlags eVisibleDamageFlags;
	};
	uint8_t bPackedIntegerCount;
	SVisibleEquipmentDamage sData[5];
};
```

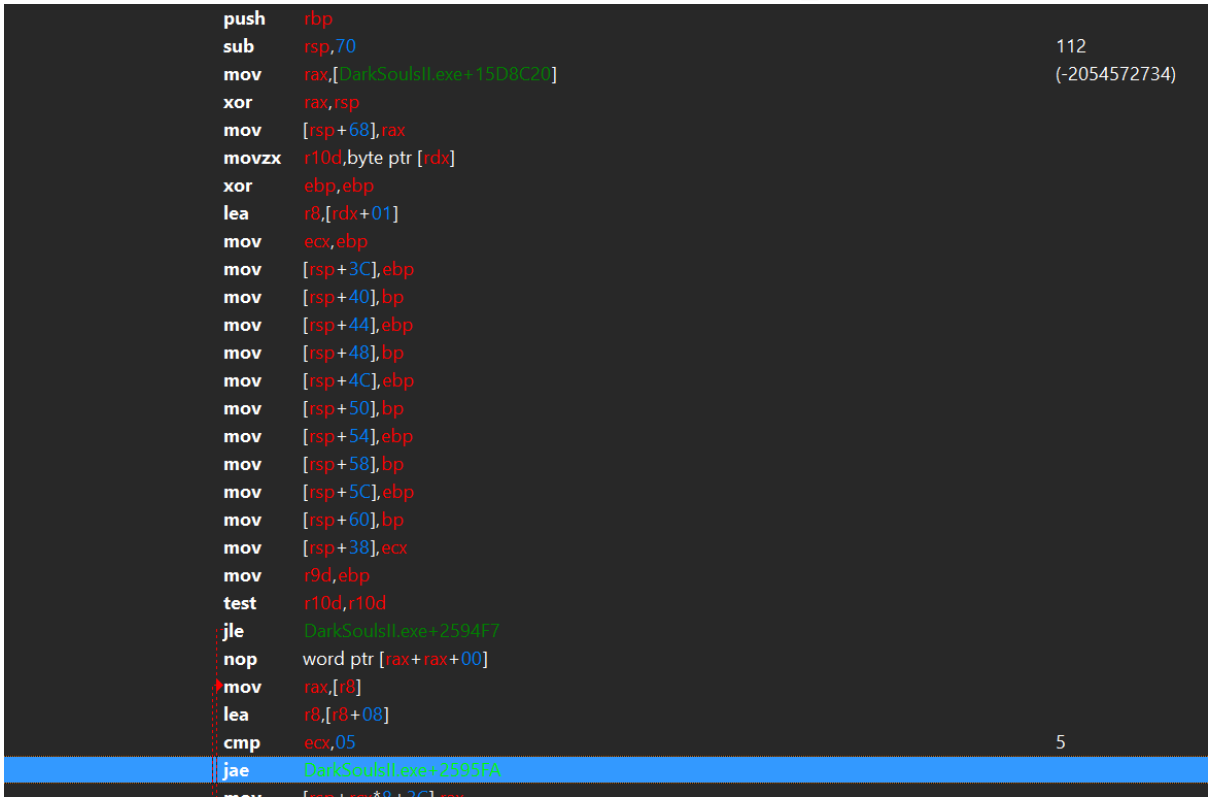
This packet updates the visuals on broken equipment between players.

Fixed vector crash

The value ‘bPackedIntegerCount’ relates to the amount of entries declared in ‘sData’. Despite the maximum amount being 5 or below, a malicious player can set this value to a value greater than 5. This causes the game engine to panic with the debug string:

“追加できる最大数を越えています(最大:%d 現在:%d)”
“N:\FRPG2_64\source\FRPG2\Title\Source\Network/Util/NvdFixedVector.h”

Below is a screenshot of the fixed vector function at fault (sub_0x259470)::



WhoID spoofing

The player can change ‘SVisibleEquipmentDamage::iPlayerWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can apply the effect of this packet to whoever they like in the session.

To address this issue, ‘SVisibleEquipmentDamage::iPlayerWhoIs’ should be strictly checked to ensure that it does not result in the PlayerIns pointer for the local player.

ChrStatusPacketReceiver

67 - “Change hollow state”

Below is how I understand that this packet data is structured:

```
struct SPacket67 {
	uint32_t iPlayerWhoIs;
	uint8_t bHollowLvl;
};
```

WhoID spoofing

The player can change ‘iPlayerWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can change the hollowing state of any other player in the session.

To address this issue, ‘‘iPlayerWhoIs’ should be removed and the game should use the SteamSessionPlayer pointer to get the PlayerIns instance of who sent the packet, and apply it to that player with no exceptions.

92 - “Change covenant”

Below is how I understand that this packet data is structured:

```
struct SPacket91 {
	uint32_t iPlayerWhoIs;
	uint8_t bNewCovenant;
};
```

WhoID spoofing

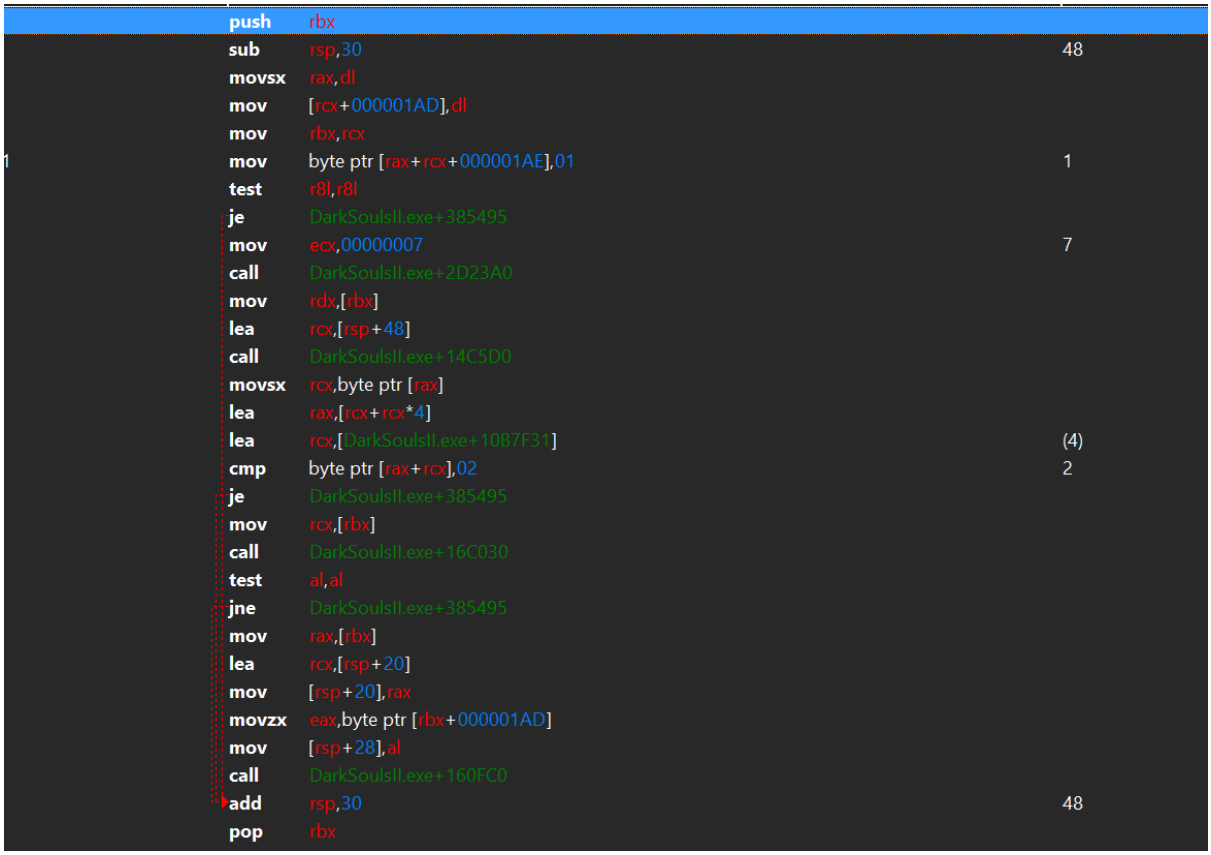
The player can change ‘iPlayerWhoIs’ to target any player in the session. As with other packets that use this ID, the ID is used to get a PlayerIns pointer for the target of the function of the packet. Using this, any player can change any other player’s covenant with this packet.

To address this issue, ‘‘iPlayerWhoIs’ should be removed and the game should use the SteamSessionPlayer pointer to get the PlayerIns instance of who sent the packet, and apply it to that player with no exceptions.

Invalid covenant IDs

The game uses the ‘bNewCovenant’ to access an array in order to set the ‘IsDiscovered’ bool for the respective covenant. There is no bounds checking here, and any value greater than 9 will result in an out of bounds write and potential crash of the application, depending how far the write is and whether it writes over any important data structures.
To fix this issue, strict bounds checking should be ran before accessing this (or any) array.

Below is a screenshot of the function which causes this out of bounds write (sub_0x385420). ‘dl’ is a byte value from ‘bNewCovenant’.



EventPacketResultCtrl

72 - “Start boss”

Sending the host home

This packet will signal that a boss battle has started. The issue with this is that it can be sent from a non-host player, to the host which will cause the host to be “sent home” to their own world. As the host is already in their world, they will crash on the loading screen.

As this packet can be used by players to send others home, I recommend this packet to be checked that it is received from the host, and disallow its use in arenas.

73 - “Host disconnect”

Sending the host home

This packet will signal that the host has left the area. The issue with this is that it can be sent from a non-host player, to the host which will cause the host to be “sent home” to their own world. As the host is already in their world, they will crash on the loading screen.

As this packet can be used by players to send others home, I recommend this packet to be checked that it is received from the host, and disallow its use in arenas.

NetNpcPhantomGenerateSync

83 - “NetNpcPhantomGenerateSync”

Below is how I understand that this packet data is structured:

```
#pragma pack(push,1)
struct SNetNPCPhantomChange {
	uint8_t bPackedIntegerCount; // > 5 will crash
	uint32_t pEntityWhoId[5];
};
#pragma pack(pop)

struct SPacket83 {
	uint16_t wSummoned;
	uint16_t wDismissed;
	SNetNPCPhantomChange sSummoned;  // Always here
	SNetNPCPhantomChange sDismissed; // Not always here, it's the sum
of the above entries plus this entry
};
```

The purpose of this packet is to synchronise NPC summons and invaders across game clients. If they occur after the player has already loaded in (otherwise they are sent with the host world data packet on connect). The parsing function is sub_0x250BF0

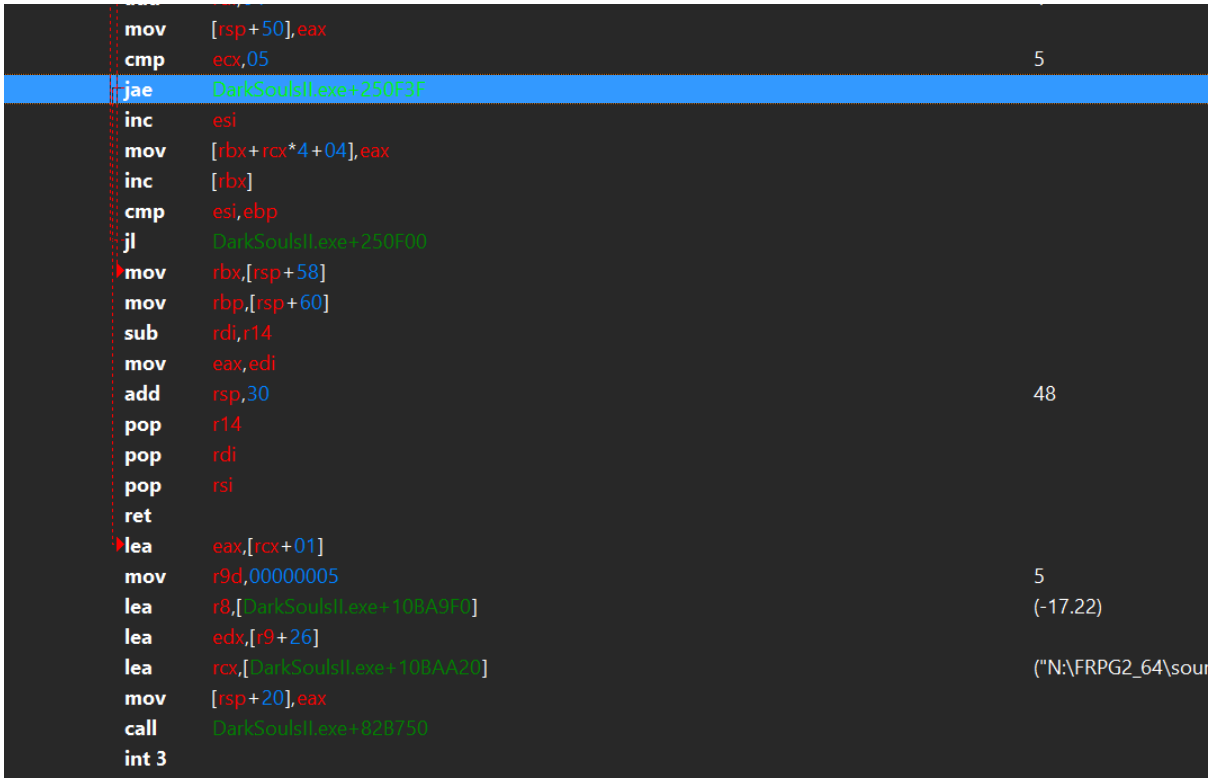
Fixed vector crash

The value ‘SNetNPCPhantomChange::bPackedIntegerCount’ relates to the amount of entries declared in ‘sSummoned’ or ‘sDismissed’. Despite the maximum amount being 5 or below, a malicious player can set this value to a value greater than 5. This causes the receiving player’s game engine to panic with the debug string:

“追加できる最大数を越えています(最大:%d 現在:%d)”
“N:\FRPG2_64\source\FRPG2\Title\Source\Network/Util/NvdFixedVector.h”

To fix this issue, I recommend ignoring invalid data like this, instead of crashing the game.

Below is a screenshot of the fixed vector function at fault (sub_250ED0):


mov [rsp+50],eax
cmp ecx,05 5
jae DarkSoulsII.exe+250F3F
inc esi
mov [rbx+rcx*4+04],eax
inc [rbx]
cmp esi,ebp
jl DarkSoulsII.exe+250F00
mov rbx,[rsp+58]
mov rbp,[rsp+60]
sub rdi,r14
mov eax,edi
add rsp,30 48
pop r14
pop rdi
pop rsi
ret
lea eax,[rcx+01]
mov r9d,00000005 5
lea r8,[DarkSoulsII.exe+10BA9F0] (-17.22)
lea edx,[r9+26]
lea rcx,[DarkSoulsII.exe+10BAA20] ("N:\FRPG2_64\sour
mov [rsp+20],eax
call DarkSoulsII.exe+82B750
int 3

NetNpcPhantomDeadSync

84 - “NetNpcPhantomDeadSync”

Below is how I understand that this packet data is structured:

```
struct SNetNPCDeath {
	uint32_t iNPCWhoID;
	uint32_t iNPCDeathReason; // 1 = Slot death, 2 = instant DC
};

#pragma pack(push,1)
struct SPacket84 {
	uint8_t bPackedIntegerCount; // > 5 will crash
	SNetNPCDeath sDeathInfo[5];
};
#pragma pack(pop)
```

The purpose of this packet is to synchronise NPC summon and invader death/disconnects while the player is in a session. The parsing function is (sub_0x250A10)

Fixed vector crash

The value ‘SNetNPCDeath::bPackedIntegerCount’ relates to the amount of entries declared in ‘sDeathInfo’. Despite the maximum amount being 5 or below, a malicious player can set this value to a value greater than 5. This causes the receiving player’s game engine to panic with the debug string:

“追加できる最大数を越えています(最大:%d 現在:%d)”
“N:\FRPG2_64\source\FRPG2\Title\Source\Network/Util/NvdFixedVector.h”

To fix this issue, I recommend ignoring invalid data like this, instead of crashing the game.

MapItemPackNetworkCtrl

86 - “Map item create”

Below is how I understand that this packet data is structured:

```
struct SItemDrop {
	uint32_t iUnk;
	uint32_t iItemID;
	float fDurability;
	uint16_t wQuantity;
	uint8_t bUpgrade;
	uint8_t bInfusion;
};

struct SPacket86 {
	uint32_t iPlayerId;
	uint16_t wItemDropType;
	uint16_t wPadding;
	SItemDrop sItem[8];
	uint8_t bAmount;
	uint8_t bIsUnknownItemType;
	uint8_t bPrismVfxColour; //"DarkSoulsII.exe"+1E3208
	uint8_t bIsGroundItem;
	float fX;
	float fZ;
	float fY;
	float fR;
	uint32_t iMapID;
};
```

This packet is sent whenever a player drops an item, and its function is to spawn that item and its contents on other game clients. It is also sent when a player drops a prism stone.

Invalid item types

The ‘wItemDropType’ entry refers to what “type” of MapItem object is created. They have the following values:

- 0 = ‘MapItemLot’ object
- 1 = ‘EnemyItemLot’ object
- 2 = ‘PlayerItemDrop’ object
- 3 = Prism stone object

Despite only these 4 values existing, and it being impossible to create types 0 and 1, this field is entirely unchecked. It is used to access an array which is *normally* checked, however the check for the array is not performed on the MapItem receiver, and the game client blindly trusts that the remote client who is sending the data is not malicious.

Using any other value will cause the game to crash.

Stack overflow

The entry ‘bAmount’ is used to tell the game how many ‘SItemDrop’ entries are in the array. Despite the maximum array size being 8, this value is unchecked and allows values of up to 255 to be entered. If this happens,

To resolve this issue, I recommend that the maximum array size is enforced when examining the ‘bAmount’ entry.

89 - “Item Pickup”

Below is how I understand that this packet data is structured:

```
struct SPacket89 {
	long long qItemBagId;
	uint8_t bUnk1;
};
```

This packet is sent by the host, when a guest player sends a request to pick up an item on the ground. The host will send back the item bag object identifier to the requesting client which will allow the pickup to occur.

Host only packet

Despite the fact that only the host can send this packet. There is no check to ensure that the sending client is the host. This means that guest players can force the host of the world to pick up items which should be impossible.

Force item pickup

Upon receiving this packet, there is no check on the local client to ensure that the item has been requested in the first place. To this end, it is possible for a malicious player to drop invalid/bannable items in the world, then send a packet 89 to players to force them to pick it up.

This is severe, as it causes innocent players to get flagged by the anti-cheat system and banned. This can be chained with other exploits (e.g. +14 invalid weapons) to corrupt save data.

To fix this, I strongly recommend that the client makes sure that it has actually attempted to pick up an item before it accepts a packet 89.

Miscellaneous

It should be noted that the above exploits are applicable both in the P2P setting (when players join each other for multiplayer activity) **and** the asynchronous online component in certain cases.

For example, ‘DeadingGhosts’ (bloodstains) and ‘WanderingGhosts’ send similar data to packet 13 (remote player data) and include fields such as ‘iSlotInfo_ActiveEquipLeft’. This means that it is possible for a malicious user to create a blood message or player ghost which will cause crashes and out-of-bounds reads on other player’s games without them needing to engage in multiplayer at all (just being connected to the server is enough).

Conclusion

Dark Souls II, for the most part, has certain checks to ensure that packets received from other players are the correct size. This is good, however as you can see the **overwhelming majority** of exploits and security vulnerabilities I’ve described in this document are caused by a lack of bounds checking in arrays.

There are very often no checks, and the game will trust other peers not to send malicious data. This is unfortunately very damaging in a game where a large portion of cheaters are now able to corrupt save data and get other players banned inappropriately.

I strongly recommend that, in addition to the fixes I have suggested, that Dark Souls II also implements a form of “Player blocking” similar to what exists in Dark Souls III and Elden Ring. This would allow players on the Steam platform to block other players and no longer join their multiplayer sessions.