Attaching Debugger to SOFTWARE-X & Single Stepping SOFTARE-X Without Crash [Guide] ~2024

So SOFTWARE-X has a bunch of stuff to make it difficult to attach a debugger. They have various ring3 methods and do their own patching of their own libraries and even their own game to ensure it is difficult. And with intrinsic CPU cycle time checks everywhere and their constant fiber based CRC32 ~30ish ~threads/fibers that look for patching anywhere it can be a daunting task to single step the game.

If you are familiar with hypervisors or KVM or QUEMU or even WINE and have an intel CPU - then you are in good shape to get around all of their counter/anti/annoying checks and research the game on your own time. Those with AMD CPUs have other projects like AetherVisor (google AetherVisor ******), but it doesn't have the capabilities and ease that many open source projects that exist for Intel extensions have. Anyways, this guide assumes you have intel CPU with intel VT-X enabled in the bios. Now, I use icebox project along with patched kernel driver Virtualbox to debug SOFTWARE-X. It is a bit annoying to use until you release you can have multiple vCPUs to install things like SOFTWARE-X and other stuff, and only requires 1 vCPU when actually do introspection work such as attaching debugger and single stepping SOFTWARE-X.

There is a branch of icebox that focuses way more on security and counter-measures, but for SOFTWARE-X this branch is not needed nor required (https://******.com/Axoosha/icebox/tr...fea234e963a510). You can go with the normal https://******.com/thalium/icebox binaries released in 2020 because building icebox can be a real pain... if you do wish to build it from scratch follow the tutorial exactly and most of all make sure you download the files with the specified hash values as written in the build guide.... that is the best advice I can give to someone who wants to build themselves without lots of headache.

Assuming you now downloaded the prebuilt files from 2020 and you are on windows 10 latest update (never tested with windows 11, but I don't see why it wouldn't work - hell - windows has their own dedicated backwards compatibility software to help run old programs :)).

Okay... So you need to enable test signing to inject the kernel drivers for VirtualBox into your kernel (or disable PatchGuard - google EFIGuard- with the branch with modifications regarding DSE control), but just going to console/terminal/command-prompt and typing:

bcdedit /set testsigining on

And then hold down shift key when you are restarting windows to bring up menu where you go to the place that allows you to have additional options on boot where you want to hit '7' for disable test signing mode or something like allow unsigned drivers to load in kernel. I forget the text it says, but it is normally key 7 on pre-windows boot menu that pops up.

Now go to icebox pre-built download's Virtualbox folder and on administrator command prompt run install.cmd. There will be two red boxes that show up warning you that an unsigned kernel driver is trying to load. Allow the two drivers to load. Then in same directory run virtualbox.exe :) This is where you go to google and type windows 10 ISO download and use windows own media center software to make you a brand new ISO file to use with Virtualbox to build a new virtual OS that will run SOFTWARE-X (remember - icebox requires 1 vCPU ONLY when you are actively utilizing it - so go nuts on vCPU count and RAM size, but remember to set vCPU # to 1 before trying to actually utilize icebox on SOFTWARE-X. Also make sure your Virtualbox CPU settings are set to paravirtualization interface = legacy and VT-x and nested paging are checked. I also have enable PAE/NX enabled, but not sure if it helps. You can have any vRAM size you want (unlike vCPU count when actually instrumenting with icebox) . with more details than provided here.

So now you have a windows 10 VM with SOFTWARE-X and SOFTWARE-Y and all that following installed and updated. You should be able to launch the game inside the VM. If you have issues with that there are VirtualBox stealth ****** projects that can help for things like changing CPU cycle counting with virtual time (TSC stuff) and other (many) variables you can edit in the VirtalBox (.vbox) file that can help hide you are running inside a VM, but I didn't need any of that when I went through this processes. SOFTWARE-X launched normally and I could play on SOFTWARE-Y as normal.

Now.... the medium difficult part - making Icebox happy, and thus making wingability as well as test_fdp.exe (example program that illustrates what Icebox can do with instrumentation and how fast it can do it - things like hidden inline hooks or hardware breakpoint hooks or hidden memory or altering the cr3 or messing with page tables or reading/writing to anywhere in virtual memory including kernel). To make icebox happy you need proper PDB files (!!! THAT ARE MADE SPECIFICALLY FOR YOUR VERSION OF WINDOWS !!!) - Linux users should have no issues because Icebox was written for Linux people I think mainly and there is a bash script that will take care of downloading all correct PDB files for you and placing them in C:\Symbols (or wherever your _NT_SYMBOL_PATH environment variable is set up to look). Go to search and type 'env' and then go to the environment variable window and add a new key called "_NT_SYMBOL_PATH" (without quotes) and make its value "srv*c:\symbols*https://msdl.microsof** ***m/download/symbols" (without quotes) and make sure C:\symbols folder really exists!

If you are not a Linux user than you need to download the basic PDB files manually for your VM Windows OS. A couple of links:

https://techcommunity.microsof*****m...er/ba-p/342969
https://******.com/kiinoo/PDB-Downlo...f910bb868b5585

...
...
...

various other public open source PDB downloaders on the internet

The PDBs you want to focus on are:

C:\Windows\System32\ntdll.dll
C:\Windows\System32\KERNELBASE. dll
C:\Windows\system32\d3d10warp.dll
C:\Windows\System32\KERNEL32.DLL
C:\Windows\system32\dwmcore.dll
C:\Windows\system32\ntoskrnl.exe
C:\Windows\System32\ci. dll
C:\Windows\System32\dbghelp.dll
C:\Windows\System32\dbgcore.dll
C:\Windows\System32\dbgeng.dll
C:\Windows\System32\hal.dll
C:\Windows\System32\kd.dll
C:\Windows\System32\kdnet.dll
C:\Windows\System32\kernel.appcore.dll
C:\Windows\System32\ole32 .dll
C:\Windows\System32\sfc.dll
C:\Windows\System32\vcruntime140.dll - Note: this might not be 140 version for you
C:\Windows\System32\wmi.dll
C:\Windows\System32\ win32k.sys
C:\Windows\System32\win32kfull.sys
C:\Windows\System32\win32kbase.sys
C:\Windows\System32\Taskmgr.exe
C:\Windows\explorer.exe

Note: I may have missed some dlls. It should tell you when trying to launch test_fdp.exe and/or wingability.exe what PDBs are missing if it doesn't have enough or the right ones to start Icebox successfully. Getting test_fdp.exe to execute (keep in mind the version released pre-built in 2020 has issues with test_fdp.exe where it blocks early on specific tests and will look frozen when ran unless you build your own test_fdp.exe to remove those ~1 -3 tests that fail). I had no problems with the pre-built wingability.exe file in the pre-built IceBox folder. Also - while you are in the icebox folder go ahead and install the python side with something like "python setup.py build && python setup.py install" from an admin command prompt.

Okay - the PDBs on Windows and getting (mainly) wingability to launch successfully and provide you a named-pipe is the MAIN/MAJOR/PROMIENT goal that you are trying to achieve to single step SOFTWARE-X. The test_fdp.exe program just shows what Icebox can do (amazing and fast things and to prove your setup is working correctly). Wingability.exe is the program that utilizes parts of that patched virtualbox kernel drivers you loaded earlier to give the user a named-pipe that you can connect windbg.exe to and debug the entire VM Windows 10 OS (if you want... I instead used IDA and disassembled a unpacked in-memory executable file of SOFTWARE-X.exe along with proper import reconstruction - and used that IDA .i64 file to connect through WinDbg.exe using kdsrv (kdsrv -t tcp:port=6666 - you can get windbg along with kdsrv.exe via Microsoft's Windows 10 Windbg or just google windows 10 debugging utilities download (maybe add keyword windbg) and install that. Add the install folder to your environment path so you can access both windbg and kdsrv via a regular admin command prompt).Also to get a well formed unpacked and import available version of SOFTWARE-X.exe I utilized x64dbg along with some plugins such as scyallahide (although the normal settings seemed to work - you need to enable the box for the debugger break-in function being patched with a 0xC3 (ret instruction) instead of 0xCC (int 3) instruction (one of SOFTWARE-X's anti-debug tricks) and of main importance is the use of scyalla for dumping (which is included with x64dbg) and also of extreme importance to recover the imports (not all of the imports - SOFTWARE-X loads dynamically special imports it wants to hide internally, but you will get most of them this way) is a ****** x64dbg plugin called SOFTWARE-Z-dump-fix (google SOFTWARE-Z-dump-fix and try to obtain a pre-built plugin for x64dbg x64 (since this guide is targeting the 64 bit version of SOFTWARE-X) or build it yourself using the latest branch found at https://******.com/fengjixuchui/Over...04a97149155fa5 .

Once you have installed the SOFTWARE-Z-dump-fix plugin and maybe scyallahide and of course scyalla (which is included automatically) - your job now is to dump a unpacked w/ imports version of SOFTWARE-X.exe while SOFTWARE-X is running - preferably while inside of an active game so everything is set up and variables are normal for a SOFTWARE-Y game - go to custom map and waht i do is make a bunch of AI bots and make sure the game is publicly advertised and launch the game on SOFTWARE-Y just like you are practicing against real AI to get better (there are other offline - no SOFTWARE-Y - methods of course which will share if details are wanted by people).

Okay - so now you have open x64dbg (ran as administrator with the SOFTWARE-Z-dump-fix plugin and scyalla) as well as you are on SOFTWARE-Y inside a game with a bunch of AI

bots. This should be the situation on your computer... Variations I am sure could work as well, but this method works since I have done it many many times. Anyways... Lets dump a good quality exe file for SOFTWARE-X.exe (the original one on your hard drive is packed and like ~68 MBs). The one we are going to dump should be around ~138 MB on disk. So... Download something like process hacker 2 program on google and install and run as admin. Inside process hacker 2 make sure SOFTWARE-Y is not running, make sure agent.exe is not running, make sure COMPANYS-SOFTWARE-X.exe is not running. All those support programs need to be killed (right click and click kill process tree inside of process hacker 2). You want SOFTWARE-X.exe all alone for this next part :)

Once the above paragraph is done, right click on SOFTWARE-X.exe and click freeze or suspend or whatever. This is now a time game - so hurry up before SOFTWARE-X.exe somehow manages to kill itself - you have like 3-5 minutes at least or maybe unlimited time... But after suspending the process all threads should be inactive state and no CPU % coming out of SOFTWARE-X.exe. Go to x64dbg and go to file and click attach - then select SOFTWARE-X.exe and hit attach. There is a logging tab in x64dbg that writes a bunch of stuff. Since you have SOFTWARE-Z-dump-fix x64dbg plugin installed you can go to the box where you type at the bottom of x64dbg and type in "SOFTWARE-ZDumpFix" or whatever it suggests is the right phrase once you type the 'O' key. Hit enter with the full plugin name correct and it will log a bunch of stuff to your logging tab. You are looking for where the IAT and the size of the IAT are located. What I do is select all and copy the contents of the logging tab and then search for 'SOFTWARE-Z' to find where the text is located. Also I forgot - you need to set up x64dbg preferences correctly. Mainly make sure all exceptions are ignored and given passthrough directly to SOFTWARE-X program and not the debugger. You do not want to break on exceptions. You want all exceptions to pass through as if no debugger is present. The SOFTWARE-Z-dump-fix plugin also takes cares of the replacement of the 0xC3 with the normal 0xCC patching so you don't really need scyallahide plugin :)

Now it is time to dump SOFTWARE-X.exe from memory along with imports. Go to plugins on top menu and hit scyalla. Select from dropdown menu SOFTWARE-X.exe. It may already be on that program in that case select a different program and then select SOFTWARE-X.exe from dropdown list. There are three main boxes for scyalla - the entry point address, the IAT address, and the IAT size. All three boxes are in hexadecimal format! No 0x or h character on the end are needed. You can get the IAT address and size by searching in the logging tab contents for what SOFTWARE-ZDumpFix printed out and you will see there are two lines back to back with the first being IAT address and second being IAT size. That takes care of two of three scyalla boxes. For the entry point or OEP or whatever box hit the ~auto-search button next to it in the scyalla GUI. There is a also a button that searches for IAT and size button. This doesn't work - use the contents that SOFTWARE-ZDumpFix provided! Once the three boxes are filled then hit the search IAT button and the big box in the middle of scyalla GUI should fill with imports from many libraries. If that happens - GREAT! If it does not and you can't recover the imports - that sucks ... but play around with it and make sure you are using the correct IAT address from the SOFTWARE-ZDumpFix text that was printed after you ran the plugin in x64dbg. It works. Maybe not the first time you try, but 99.99% working by the 2nd or 3rd try of clicking the IAT search button with the 3 boxes filled in correctly. If you can't get it to work and give you a bunch of imports then respond to this thread, and maybe I can help.

Now - you are in Windows 10 with x64dbg attached to a suspended version of SOFTWARE-X.exe where you were previously on SOFTWARE-Y playing against AI. You have ran the SOFTWARE-ZDumpFix plugin after attaching the debugger to SOFTWARE-X.exe and got the printed output from the logging tab of x64dbg. You have placed the IAT information in 2 of the boxes in scyalla GUI and the OEP/entry function address box got auto-filled with the other scyalla button (or you can just use processes hacker 2 and click on SOFTWARE-X.exe process and go to exports and you will see a start export or something similar with an address. SOFTWARE-X.exe by default has random startup addresses relocation base address - you can use detect-it-easy (google it) to make a custom version beforehand of SOFTWARE-X.exe packed exe file inside SOFTWARE-X home folder - .\Versions\BaseXXXXX\SOFTWARE-X.exe - inside detect-it-easy go to the flags box and remove randomized based and the other flag that suggests by context some kind of random base value - you also need to unclick he "Read-Only" box in the upper right corner of detect-it-easy to make these 2 PE header flag changes). If you go about doing this you can rebase your application anywhere you want in virtual memory to say base address of 0 or say normal base address of 0x1400000.... which it will default to after you remove the two flags and detect-it- easy will automatically make the edited version and also backup automatically SOFTWARE-X.exe.bak inside the same folder for you.

Anyways - that last paragraph is for those wanting specific base address that will match with your IDA pro disassembled .i64 file. This helps incredibly when debugging via IDA Pro --> Windbg --> kdsrv --> named-pipe such that your rebased SOFTWARE-X IDA Pro memory range corresponds to the memory range that it is actually running at inside its virtual memory inside of VirtualBox (IceBox style :)) - if you don't have IDA and the actual SOFTWARE-X.exe that runs inside of VirtualBox sharing the same base address than you will need to rebase your IDA setup which is a pain, but is definitely a possible solution to avoid all this PE header flag removal stuff talked about. But you will very quickly get annoyed because whenever you run SOFTWARE-X.exe inside the Virtualbox it will have a random base address and you will need to rebase your IDA each time else the virtual memory addresses won't line up and that is a big problem for say setting breakpoints on the TLS callbacks or the wWinMain entry or say on a virtual memory address that you need to patch as SOFTWARE-X is starting such that CRC32 checks are removed and do not occur and detects your patch for maphack which I will provide a signature for as an example for a "Hello World" type of thing for debugging/single-stepping/break pointing and patching of SOFTWARE-X.exe.

Okay... Now I got side tracked, but we are back in the scenario where you are using scyalla plugin for x64dbg and are about to dump a ~138 MB version of SOFTWARE-X.exe unpacked and with imports executable file to your hard drive. There are 3 other buttons near the right corner of scyalla GUI. One is like dump, one is like fix dump, and there is third button I don't recall what it says. But start with the dump exe button and then figure out what is the proper order for the next two buttons that each have you point to the path on hard drive where you have just dumped an unpacked version of SOFTWARE-X.exe. Great! Now you can stop x64dbg debugger with debugging menu such as kill target or detach from process (either way SOFTWARE-X.exe is going to crash afterwards - you can try to resume the process if you do all the x64dbg stuff quickly enough and SOFTWARE-X.exe will continue to run as if nothing happened if you do it quick enough - well no - .... - SOFTWARE-X.exe will most likely crash any ~most scenarios after you have attached x64dbg debugger to it even though it is suspended You just don. 't want it to crash before you have dumped your unpacked and with import rebuilt version of a ~138 MB SOFTWARE-X.exe to your hard drive to later provide to IDA Pro for disassembly and later on debugging with. and just use WinDbg if you are more comfortable in that environment. There is also WinDbgX (aka WinDbg Preview) which comes with better GUI and most of all TTD.exe (time-traveling-debugger) - this can later be used to make big trace files of specific aspects of SOFTWARE-X.exe when it is running and then replay in a timeless fashion inside of IDA Pro, and it doesn't ever try to attach to SOFTWARE-X.exe so one can utilize TTD.exe without any anti-protection steps to make trace files of SOFTWARE-X.exe operations. These files are BIG and you need to convert them to a format that IDA Pro can understand. So utilize IDA Pro plugin like tenet or lighthouse or various other tracing plugins (you may have to convert the format of the TTD.exe trace to another format for it to work with IDA Pro plugin or even IDA Pro tracing internals... Another tangent ... Sorry and apologies.

Now - you have a dumped unpacked w/ imports version of SOFTWARE-X.exe (ideally with a non-randomized base address, but NOT required because you can later always rebase your IDA Pro environment to where ever SOFTWARE-X loaded in virtual memory). Great! First thing is first - if you are using IDA Pro to debug with (which I use because I am not as familiar with Windbg commands) you need to disassemble the newly created exe file that you dumped in IDA Pro and allow it to finish analyzing. Also try to utilize Lumina server for adding signatures and symbols. Once IDA Pro is done analyzing it is finally time to single step and debug and breakpoint SOFTWARE-X without any awareness by it such as all those timing checks and heap debugging checks and their vector exception handler and their normal exception handler (which they interestingly patch the library function for... they also patch dbghelp.dll, ntdll.dll, SOFTWARE-X.exe, kernel32.dll, and others). Okay... So open a command prompt. No, open two admin command prompts. You can run ida pro on aa remote computer or on the same computer as the Icebox modified version of VirtualBox. Your goal is to utilize wingability.exe example with the correct name of your VirtualBox VM name and the proper syntax that it will tell you if you just run wingability.exe without any arguments. You want to utilize the FDP method of wingability since it is super fast and great at break pointing and making inline invisible patches, as well as providing you a named-pipe that you can hand-off to WinDbg so it knows what it needs to attach the kernel debugger to.

Lets assume you are doing this all on the same computer, and you want to utilize IDA Pro instead of just using WinDbg to debug SOFTWARE-X. In one command window type "kdsrv -t tcp:port=6666" or whatever port number you want above a few thousand because the lower ports are reserved for specific application such as HTTP being port 80. Hit enter on that command prompt and you now have kdsrv running (verify with process hacker that it is running). You may also need to provide an IP address to kdsrv such as 127.0.0.1 for when running everything on one computer, but most likely not needed depending on your network setup. Now you need to combine all these ingredients into a connection string that IDA Pro will feed to WinDbg and ultimately allow you to have IDA Pro attach a kernel debugger to your Windows 10 VirtualBox VM running SOFTWARE-X. If all your environment variables are set correctly (namely _NT_SYMBOL_PATH along with C:\Symbols and those PDB files you obtained earlier to help get Icebox working correctly) IDA Pro will automatically download countless PDB files for you to make your debugging life more convenient due to symbol information. Also in IDA Pro when configuring the Windbg parameters I always completely removed all exceptions by deleting them from the list such that it is blank, I click the boxes that do stack auto-reconstruction, auto-PDB loading, Auto-hidden int3s for breakpoints (this doesn't include the 4 hardware breakpoints), and there was one other obvious box that I checked because its text description just made sense to have enabled... Also I remove (normally) all the auto-break-in points such as Debugger Break-in; new process break-in; new thread break-in; debug break-ins, and all the rest so none of them are checked. Also make sure that IDA Pro WinDbg is configured to debug the full kernel debug mode (not those other less intrusive methods that are options).

Okay... In that other command admin prompt that you didn't run kdsrv in you now need to run wingability.exe in. This is the final and sorta hard part to get working correctly - mainly due to not having the proper PDB files already downloaded and inside of C:\Symbols or wherever your _NT_SYMBOL_PATH environment variable says to look. But, if you have all the correct and 100% aligned to your version of Windows and the PDB files listed above matches that version you shouldn't have a issue when running wingability.exe. You know it worked when it provides you a named-pipe and doesn't auto-exit after running it. My syntax (based on what I call my VirtualBox VM) is:

winbagility.exe \\.\pipe\client FDP win10

Note: win10 is the name of my Virtualbox VM
Note: winbagility can sometimes look like it worked and ran fine, but make sure the fields on the console like IopNumTriageDumpDataBlocks, ..., v_KDBG, v_KernBase, v_KPRCB[0], and v_KPCR[0] all have good looking kernel virtual addresses assigned to them and not something like 0x00. The kernel is in the upper part of virtual memory so the addresses will look like 0xFFFFF803076000B20 as an example. Anyways, it is rare that winbagility does bad addressing, but it is something to check for. If all looks good, at the bottom of the output it will say:

"[Main] NamedPipe \\.\pipe\client created ! Waiting connection..."

Note: if it says something like that and has not auto-exited then you are essentially near the end of this adventure :)
Note: if you are having issues it is most likely due to improper PDB files for your VM version of Windows 10. You need to correct and properly versioned and right PDBs for wingability and Icebox to function. So focus on getting the right PDBs. You can brute-force it and download every PDB inside C:\Windows for example. Or you can play around with the Icebox python you installed, and try to discover which PDB file it gets hung up on not having and obtaining that PDB file and do it again and again until it has all the PDBs it wants to function correctly.

Well almost - there - in IDA Pro when you want to debug SOFTWARE-X - and after you have configured IDA Pro to utilize WinDbg as debugger and as a full kernel debugger with out any exceptions on the list and those check boxes we talked about set up already - you are ready to go to Debuggger menu on top and then Process Options. Inside here you want the Application box to point to your dumped SOFTWARE-X.exe file, same with Input File box, Directory Box you can set to C:\Program Files (x86)\SOFTWARE-X\Support64\ directory (or leave blank.. .maybe...); Parameters box can be empty if you are attaching to an already running SOFTWARE-X inside your VirtualBox VM (if not already running you can add things here like -run "%PATH_TO_MAP_NAME%" -sso .... I don't remember... Just leave the Parameters box empty and it will work even if SOFTWARE-X is not yet running inside the VM. Also if some user wants more information about how to launch SOFTWARE-X.exe from command line without going through SOFTWARE-X-Switcher_x64.exe let me know.) And finally the long talk about Connection string box. My connection string is the following:

kdsrv:server=@{tcp:port=6666,server=127.0.0.1},tra ns=@{com:port=\\.\pipe\client,pipe,baud=115200,res ets=0,reconnect}

Note: you have already run kdsrv and you have already ran and is also running (just like kdsrv) wingabilty (I keep spelling that wrong - sorry) in the other admin command prompt window. Notice my named-pipe matches between the connection string and what wingability printed out on its' last output line. Notice the localhost (127.0.0.1) IP address that you already setup with kdsrv (this could be a foreign computer on your private network as well that is running the Icebox version of VirtualBox). Finally check the box in IDA "Save network settings as default" and click on the "OK" button. Finally go to the Debugger menu in IDA Pro at the top and click "Attach to process"! And wait... If all goes good IDA Pro will start activity such as downloading PDBs automatically and you will finally end up with a VirtualBox VM running SOFTWARE-X which is completely frozen (aka 0% CPU activity) and IDA Pro will have broken into the VM as the kernel debugger at whatever point the OS was at whether it be in userspace or kernel space when the debugger finally attached and broke in to the VM. You can single step, set breakpoints, or anything else now. I would hit F9 to resume the VM so it isn't frozen and check that SOFTWARE-X is completely unaware of what just happened and is still running as you left it. If you have set up IDA Pro virtual memory base address to align with the base addresses in virtual memory of SOFTWARE-X inside the VM - well then - you can quickly set a hardware breakpoint somewhere such as on the TLS callback #0 or anywhere inside the entire range of memory and sections that SOFTWARE-X occupies whether it be data read/write or code execution breakpoint. All you have to do is start SOFTWARE-X or just allow it to continue running (hit F9 inside IDA Pro) and when the 1 vCPU hits your hardware breakpoint IDA Pro will freeze the VM and stop code execution inside IDA Pro exactly at your breakpoint. From there you have all the registers filled, the stack trace filled, the assembly filled, and can hit say F7 inside IDA Pro to single step 1 instruction. Or you can place a breakpoint on the string reference "devmoderation" or "devmod" (I forget) and launch SOFTWARE-X inside the VM and it will quickly hit the devmod breakpoint (data read/write - probably more so write than read -

hardware breakpoint) and you can alter its value from 0 to 1 and watch the startup behavior of SOFTWARE-X change from normal to showing a new dialog box you 99.9999999% have never seen before.

I told you I would provide you with a signature to get you started with a simple basic maphack. I keep my word (Credits go to Person-X for this basic "Hello World" maphack offset):

Note: all credits go to Person-X for releasing enough information years ago to find this maphack poistion and patching details, and it still works today! Thank you Person-X for trying to push the community forward today, and even years ago! You were the only one to release anything like this, and you cracked VCMH, and you wrote about Warden and its' checks and behavior; and much more! Thank you! ALl credits to Person-X for this maphack!

Signature = 41 88 B4 ?? ?? ?? ?? ?? 40 88 B7 ?? ?? ?? ?? E9?? ?? ?? ?? 48 8D 4C 24 ??

...
...
...

```
.text:0000000141F11386 loc_141F11386: ; CODE XREF: SOFTWARE-X_Person-X_MapHack_Function+C1 ↑ j
.text:0000000141F11386 33 DB xor ebx, ebx
.text:0000000141F11388 4C 89 A4 24 88 00 00 00 mov [rsp+0B8h+var_30], r1 2
.text:0000000141F11390 89 5C 24 20 mov [ rsp+0B8h+var_98], ebx
.text:0000000141F11394 40 88 AC 24 C0 00 00 00 mov byte ptr [rsp+0B8h+arg_0], bpl
.text:0000000141F1139C 4C 89 B4 24 80 00 00 00 mov [rsp+0B8h+var_38], r14
.text:0000000141F113A4 C7 84 24 D8 00 00 00 00 00 00+ mov [rsp+0B8h+ar g_18], 0
.text: 0000000141F113A4 00
.text:0000000141F113AF E8 4C 9E C6 FE call SOFTWARE-X_API_MapHack_Related_DATA_BYTE_Patch_Function ; 3 3 <--- you want the value referenced in this function to be equal to 2
.text:0000000141F113B4 84 C0 test al, al
.text:0000000141F113B6 0F 84 A9 00 00 00 jz loc_141F11465
.text:0000000141F113BC 41 B0 01 mov r8b , 1
.text:0000000141F113BF 41 0F B6 D7 movzx edx, r15b
.text:0000000141F113C3 48 8B CF mov rcx, rdi
.text:0000000141F113C6 E8 85 F3 FE FF call sub_141F00750 ; 8 8 <--- you want to replace this with mov al,0Bh with ~2-3 nops so mov esi,eax is the following intact instruction
.text:0000000141F113CB 8B F0 mov esi, eax
.text:0000000141F113CD 41 80 FF 10 cmp r15b, 10h
.text:0000000141F113D1 0F 84 82 00 00 00 jz loc_141F11459
.text:0000000141F113D7 45 0F B6 F7 movzx r14d, r15b
.text:0000000141F11 3DB 41 0F B6 8C 3E D8 08 00 00 movzx ecx, byte ptr [r14+rdi+8D8h]
.text:0000000141F113E4 83 C1 FC add ecx, 0FFFFFFFCh
.text:0000000141F113E7 83 F9 01 cmp ecx, 1
.text:0000000141F113EA 77 3E ja short loc_141F1142A
.text:0000 000141F113EC 83 F8 05 cmp eax, 5
.text:0000000141F113EF 7E 39 jle short loc_141F1142A
.text:0000000141F113F1 41 0F B6 D7 movzx edx, r15b
.text:0000000141F113F5 48 8B CF mov rcx, rdi
.text:0000000141F113F8 E8 43 7B FF FF call sub_141F08F40 ; 3 3
.text:0000000141F113FD 84 C0 test al, al
.text:0000000141F113FF 74 29 jz short loc_141F1142A
.text:0000000141F11401 41 0F B6 CF movzx ecx, r15b
.text:00000 00141F11405 41 BC 01 00 00 00 mov r12d, 1
.text: 0000000141F1140B 41 D3 E4 shl r12d, cl
.text:0000000141F1140E 44 09 A4 24 D8 00 00 00 or [rsp+0B8h+arg_18], r12d
.text:0000000141F11416 41 88 B4 3E D8 08 00 00 mov [r14+rdi+8D8h], sil
.text:0000000141F1141E 40 88 B7 E8 08 00 00 mov [rdi+ 8E8h], sil
.text:0000000141F11425 E9 13 01 00 00 jmp loc_141F1153D
.text:0000000141F1142A ; ---------------------------------------------------- -------------------------
.text:0000000141F1142A
.text:0000000141F1142A loc_141F1142A: ; CODE XREF: SOFTWARE-X_Person-X_MapHack_Function+12A ↑ j
.text:0000000141F1142A ; SOFTWARE-X_Person-X_MapHack_Function+12F ↑ j ...
.text:0000000141F1142A 40 84 ED test bpl, bpl
.text:0000000141F1142D 74 22 jz short loc_141F11451
.text:0000000141F1142F 41 80 BC 3E D8 08 00 00 09 cmp byte ptr [r14+rdi+8D8h ], 9
.text:0000000141F11438 72 17 jb short loc_141F11451
.text:0000000141F1143A 8D 46 FE lea eax, [rsi-2]
.text:0000000141F1143D 83 F8 03 cmp eax, 3
.text:0000000141F11440 77 0F ja short loc_141F11451
.text:0000000141F11442 41 0F B6 CF movzx ecx, r15b
.text:0000000141F11446 BB 01 00 00 00 mov ebx, 1
.text:0000000141F1144B D3 E3 shl ebx, cl
.text:000 0000141F1144D 89 5C 24 20 mov [rsp+0B8h+var_98 ], ebx
.text:0000000141F11451
.text:0000000141F11451 loc_141F11451: ; CODE XREF: SOFTWARE-X_Person-X_MapHack_Function+16D ↑ j
.text:0000000141F11451 ; SOFTWARE-X_Person-X_MapHack_Function+178 ↑ j ...
.text:0000000141F11451 41 88 B4 3E D8 08 00 00 mov [r14+rdi+8D8h], sil <--- where the signature brings you. This sil is ideally altered by that function labeled above where you patch it to return a different value (my memory says 0x0B?)
.text:0000000141F11459
.text:0000000141F11459 loc_141F11459: ; CODE XREF: SOFTWARE-X_Person-X_MapHack_Function+111 ↑ j
.text:0000000141F11459 40 88 B7 E8 08 00 00 mov [rdi+8E8h], sil
.text:0000000141F11460 E9 D8 00 00 00 jmp loc_141F 1153D
.text:0000000141F11465 ; ---------------------------------------------------- -------------------------
.text:0000000141F11465
.text:0000000141F11465 loc_141F11465: ; CODE XREF: SOFTWARE-X_Person-X_MapHack_Function+F6 ↑ j
.text:0000000141F11465 48 8D 4C 24 24 lea rcx, [rsp+0B8h+var_94]
.text:0000000141F1146A E8 11 AF C6 FE call sub_140B7C 380; 10 10
.text:0000000141F1146F 44 0F B6 AC 24 C0 00 00 00 movzx r13d, byte ptr [rsp+0B8h+arg_0]
.text:0000000141F11478 4C 8D B7 D8 08 00 00 lea r14, di+8D8h]
.text:0000000141F1147F 40 32 F6 xor sil, sil
```

...
...
...

Note: if the maphack doesn't suddenly work after hitting F9 inside if IDA Pro you may have to hit F10 inside SOFTWARE-X and say go to Window mode or go to Fullscreen mode to get the graphics engine to reset, but you probably won' t need to do that step. But, do it if the maphack is obviously not working, and it should work if the instructions were correctly followed in the comments above on the assembly lines saying what to edit data wise and what to edit in the .text section - replacing call sub_141F00750 with mov al,0Bh and nop, nop, nop (don't remember how many nops, but you want that next instruction that was originally there intact.

After you made the changes to the code as referenced in the "<--- -" comments on some of the assembly lines suddenly you will see all enemies on minimap and full map, although this particular method is kind of a darkish overlay fog of war maphack. There are 100% better methods for patching different kinds of map hacks such as say you want to see the view from your enemies camera or you want all player camera views all at once or you just don't want it to be so dark. But, those are things for you to discover with your new capability of debugging. SOFTWARE-X along with seamless break-pointing, single-stepping, tracing, etc... :)

Note: a trick to not at first have to bother with the CRC32 integrity checks if you are patching inline .text section is to place a hardware breakpoint for read only on the previous 4096 page before the one you are patching on and that way you will hit that read hardware breakpoint 1 page before the page you have edited assembly instructions. This gives you the ability to put the patches back to normal before the CRC32 fiber/thread can do its' calculation. In-fact, I believe the opcode for CRC32 does 256 or 512 bytes at a time so you don't even need to place that read hardware breakpoint 4 kB before your patches, but doing so you know you are probably safe even though SOFTWARE-X has ~30ish of these fibers constantly checking for inconsistencies in the .text sections, but each one goes in sequential byte order (you could have really bad luck and have 1 of those fibers just happen to start off CRCing where you patched, but that is supremely Unlikely!) Anyways, this is just a method to avoid CRC32 checks before you have the time to discover ways to turn off the CRC32 checking (various methods from not letting some of those fiber to start, to doing a trampoline hook on each CRC32 opcode and have it point to the original unpatched memory for it to calculate CRC32 values with (look-up ferib.dev's works - https://ferib.dev/blog.php?l=post/By...egrity_Checks; https://web.archive.org/web/20210416...egrity_Checks; https://web.archive.org/web/20210504...ib/D2R-Offlinefor reference because SOFTWARE-X x64 still uses the same AMD64 CRC32 opcode for integrity calculations); or discover your own method to trick the initialization of the CRC32 stuff to believe that it has already occurred; or patch the CRC32 look-up tables that you fill with your custom CRC32 hashes for the places you patch; many ways... Play around, mess variables up, look at behavior, study the assembly, study the back-in-time tracing, study altering development and debug variables, study and research all the cheats and development cheats and public cheats and actor cheats and test mode cheats and non-public cheats (ie, look-up "charges" cheat and look for what other cheat code is right after it :)).

This is a super long post. There is probably a easier method (I am sure of that) to single step and debug without crashing and breakpoint SOFTWARE-X 64 bit, but this method works, has worked for years, and will probably work for years to come (unless COMPANYS-SOFTWARE-X notices this post, and takes some kind of action - in which case - don't use icebox anymore - use one of the countless and more powerful instrumentation hypervisors out there for Intel processors such as VivienneVMM or HyperDbg or Zero-Tang/NoirVisor; or various others. You could also utilize KVM and/or QUEMU say with KVMs' hidden inline patching configuration.

Very long post. If you can't get it working feel free to ask for help. but once you get it working you take a snapshot and you are good to go from there on out. And this method works with various other games and anti-cheats as well.

I plan on releasing quite a bit more in the short future regarding research I have done into SOFTWARE-X. I will l probably let the information such as auto-victory hack or tie-hack or auto-automations or production panel or how to auto- decrypt the entire classes/structures for every single object in your SOFTWARE-X active game which includes everything from you and your allies to map objects to player observers/referees to all enemies and their units and buildings; in-game both team chat communications; auto-nydus warning (along with tons of other built-in warnings); even show how to obtain decrypted and structured payloads of information directly from SOFTWARE-X in the millions of bytes at once giving you everything from visibility matrices for each player to full pixel renderer layers to in-place camera lock so replay doesn't show you off looking at your enemies in the fog of war which isn't fog for you :) to custom CRC32 bypasses; and even I may add post talking about how to make SOFTWARE-X forbid itself from crashing/crashing /w minidump/fastfail/... even though it knows something is wrong with its validators or CRC32 fibers or essentially whatever. I may also release a private maphack with production panel hack open source on this fourm along with idapython scripts to auto-find for you the new offsets for new game patches/releases. I probably won't release my holy grail which is a completely different technique to obtain ~99.99% of everything you could possibly want out of SOFTWARE-X in a decrypted form and auto-structured for you already. But, who knows - I may release the full working source for that as well, but not the idapython signature offset finding script because that would just be too easy for people and a bit hard for COMPANYS-SOFTWARE-X to easily fix/patch (the method that is). Anyways, in a short time to come I will release more stuff on this forum related to SOFTWARE-X 64 bit.

Feel free to ask questions, and I will try to respond the best I can to help.