Malicious Entity Injection README

Malicious Entity Injection 101: What is MEI?

MEI, or Malicious Entity Injection is the process of loading malicious entities including visuals and executable code in to a game engine based application at run time to alter game play or the utility of the application. An example of this would be to take a Mixed Reality app used on an HMD, and use MEI to replace a model or execute code to overlay a logo over a face to conceal someone's identity, steal their data (including location) that's accessible through the application, or other nastiness through manipulating the user, such as social engineering a person operating a car with an HMD on in to causing an accident. This is hardware agnostic, and while demonstrated in Unity3D, it's not beyond reason to believe Unreal Engine and other engines such as Godot are vulnerable as well. I just haven't implemented in those for this PoC. Any engine that allows arbitrary content loading and attaching visual scripts or other content that execute code to the content should be vulnerable.

The thing to remember about MEI is while code execution is bad, the bigger threat is what you can manipulate the user in to doing. Luckily with current hardware this is kind of far fetched, however with more advanced software and hardware in development, luckily this is a chance to catch it early before this becomes a real issue outside the realm of academia.

In this example, due to lack of content validation of what's being loaded in Unity3D, it is possible to use a specially crafted asset bundle to inject an entity to attack the user of an X Reality device. While in the past this would have been minor due to either being on a computer screen or a lack of the ability to inject behavior, with the rise of XR HMDs combined with the rise of visual scripting that can be attached to entities and saved/loaded in unverified content, this leads to the potential of code execution that can be used to manipulate a user in the real world with real world consequences. When combined with the ability of certain visual scripting systems to use .net DLL reflection to load every function from all DLLs loaded, you get access to a lot of functionality, including rendering, GPS data, execution of applications, and sending/receiving web requests. There is even the potential to pop shell using certain visual scripting systems. Another way to think of this is loading arbitrary javascript in a browser. While we can't exactly fix that now, XR is young and can be fixed early.

In this example, I will demonstrate on a 2D screen, however this can be easily ported to XR devices. The rationale for being called "Malicious Entity Injection" instead of something like "Visual Script Injection" is because the entity can load whatever the hell it wants, and unsigned content being loaded is bad, but this specific vector is through Entities. For the injection part, it's both an reference to DLL injection which this is similar to, and because frankly it was the first thing I thought of for how to name it and out of fluke the acronym turned out to be the same as my Overwatch Mei-n.

PoC README:
The instructions on how to run the actual attack are as follows. I've sent a zip file to the provided sharepoint. Inside the Zip file is a basic example, the directory structure is as follows:

mei-attacker-POC [Put this in the Kali VM]
MEI-Test [the "app" being attacked]
MEI-Test/attacker/poc/Meilicious-asset-bundle [the malicious bundle]
MEI-Test/attacker/poc/MEI-POC [the target]

To run the attack, start a Kali VM with a bridged network adapter. Make sure to install bettercap, as this is the tool used to execute the attack. As reported in the referred to case, running bettercap with the scripted PoC was what was causing the BSOD that seems to no longer work (when I ran the script, the host BSOD'd). Pull the **mei-attacker-POC** folder in to the VM and modify the IP addresses in **mei-http-basic.sh** to the host and gateway as needed.

The actual asset bundles are the following:

Legitimate:

Scene showing bundle source:
MEI-Test/attacker/poc/MEI-POC/Assets/MEI/Scenes/MEI-AttackDev.unity

Bundle:
MEI-Test/attacker/poc/MEI-POC/AssetBundles/StandaloneWindows/unitychanfriend.unity3d

Malicious:
mei-attacker-poc/basic/meilicious-unitychanfriend.unity3d
Source (openable in Unity Editor) is at: MEI-Test/attacker/poc/Meilicious-asset-bundle

The meilicious bundle is the attacker bundle in the asset bundle. poc.mei.ninja (referred to in the abmanger in the testbed scene, under bundleurl in the inspector) is my domain that hosts the legitimate assetbundle, this can be changed as appropriate. Run the scene in the unity editor once to see legitimate behavior, and then stop it, and start bettercap with the script provided as mei-attacker-poc/basic/mei-http-basic.sh in the VM. Restart the scene and it should replace the bundle using MITM. Note that this example relies on HTTP rather than HTTPS and HTTPS will foil it, but that's outside the scope of what this demo is trying to prove. Frankly everyone should be using HTTPS, but this isn't about HTTP/HTTPS and is about the actual asset bundle replacement, MITM is just the easy PoC vector. The concept behind this is about faking and loading unverified and malicious content, rather than the MITM, hardware, or even the engine itself.

How did I come up with this?

I was researching asset bundles for Unity3D as a means for supporting customization in a project. In the process of researching Unity's documentation on asset bundles, I found the loading remote asset bundles process involved a function that pulled from HTTP (but could be configured for HTTPS, which while hopefully everyone going forward will do, and would thwart this PoC, is going to get in the way of basic POC, so I'll stick with HTTP for now), but had absolutely zero validation involved. It blindly trusted the asset bundle downloaded. Admittedly some of the functions had CRC32 as optional verification, however let's be real about this, CRC32 is worthless and has no security value at all.

At first, I thought, meh, so what? Then I realized ***what could possibly go wrong***. Beyond specially crafted asset bundles designed to exploit the application through normal means, what if I could load a script in the asset bundle? What if this could lead to code execution? At this point, I started reading, more and more. I found out that Unity3D doesn't support including C#, JavaScript, or Boo (the three languages supported out of the box, well, boo got removed and JS is in the process of removal, so I guess just C# now, but meh), no scripts in asset bundles. OK, so we're somewhat good, that's a good thing, right? I mean, you can't load scripts, so you swap with a model, so what? Yeah it could cause some issues in some situations, but… I walked away from it for a while thinking "well, it looks like Unity designed this somewhat decent at least, not a huge problem after all".

After a short break, I came back to the asset bundle stuff and realized something "wait, what if I used visual scripting assets, do those count, would those get blocked out as scripts?". At that point I was thinking "OK, I have local, but if you have shell, they're already screwed." Then I realized you can load assetbundles from the web and stream them from memory. At this point, I needed to test, so I pulled up the editor and tried it out; it worked. I also had the realization that just swapping models and assets could get someone hurt in the right situations.

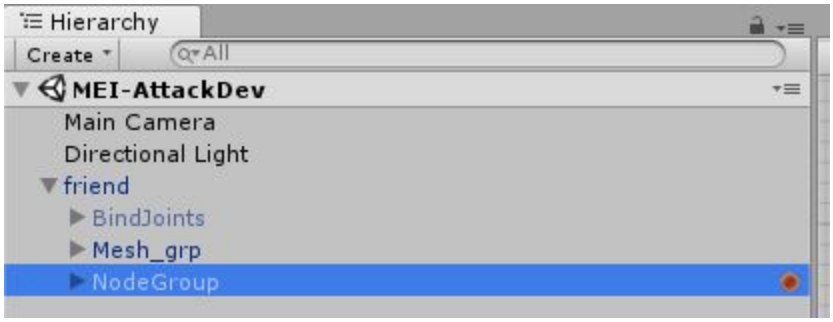
Phase 0: Why does this work?

Before I demonstrate the actual process to create and exploit the malicious payload and the vulnerable way of loading the content, I'd like to point out how and why this works. The short and sweet of it is that because Unity3D loads asset bundles raw without verifying content, as long as it has the right "interfaces", or more accurately, the right entities in the right named asset bundle, anything nested in those entities is also loaded. With this you could either nest malicious functionality in what should be loaded and appears to be a legitimate asset, or you can outright replace the legit asset with a malicious one. Unity3D and game engines in general only look for the name/path and if that checks out it loads without verification of actual integrity of the content itself. This means that entities, if malicious and with the right entry points, can be loaded with malicious content in place of or in addition to real content. The closest relation I can think of to this is a new form of DLL injection targeting assets in gaming applications such as VR/AR that allow execution of code that is in the content/asset files. Note for this example I'm

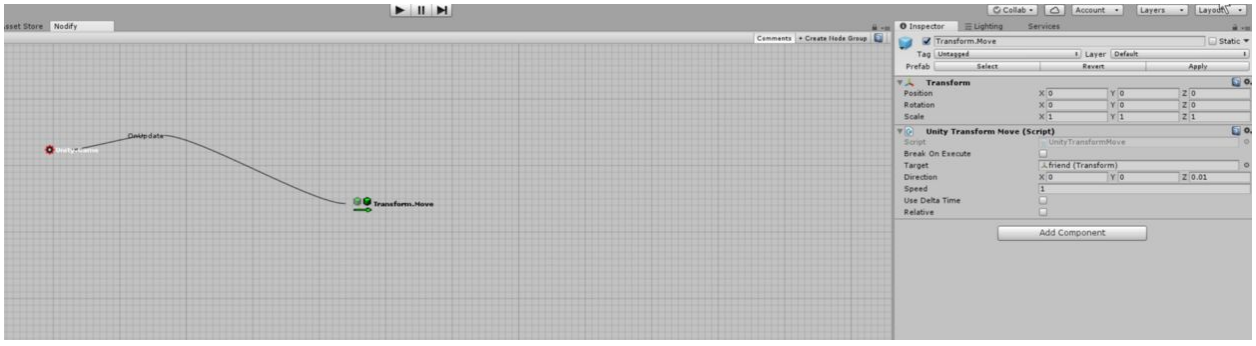
doing this over HTTP, HTTPS would thwart the man in the middle attack, but this is an academic example of what can be done, rather than a real world best practices situation which would hopefully come into play. Remember that this PoC is not demonstrating the vector but rather the payload. For certain attackers which could easily want to exploit something like this for espionage or warfare purposes, HTTPS means nothing and they can easily just circumvent it with a rogue CA, maybe one owned by the Iranian, Chinese, or North Korean governments, or with whatever security appliance is on the network that allows DPI and has a custom CA for working around HTTPS.

Phase 1: the content creation

For this I'm going to use the basic asset bundle loading example from Unity3D while loading from the web as a target. The code for the app is on the sharepoint, so I won't go over that, instead I'm going to focus on creating the valid and malicious content itself.
Here we create the default content, it's a standard asset bundle, with code right out of Unity3D tutorials and documentation. The only difference is using Nodify as a tool to enable a flow to be attached on the entity. This is what allows code execution. There are other tools that include dll reflection that allow much more functionality and also work with this attack path, such as Bolt, however I'm keeping this with non commercial assets for this example so I can share code.



The highlighted NodeGroup is where the Nodify script is attached. "friend", the name of the entity is what gets spoofed in the attack.



The Nodify flow of the valid file. This example just does a Transform.Move to demonstrate basic behaviors can be injected. More advanced PoCs could easily send REST requests, query GPS data, and more.

Note the point where it asks what the name of the content in the asset bundle and the name of the asset bundle are. That's all it looks for in loading, and that is exactly what we are exploiting.

If it matches the entity name and the asset bundle file name, it pulls a honey badger and loads it. Note that in order to find the vulnerable entry point, all you have to do is pull up the assembly in the Unity app in production in dotnetpeek and start looking for assetbundle handlers in there.

Note where it says unitychanfriend next to AssetBundle. That's the name of the asset bundle. This is what you're going to look for to spoof in your attack for the actual file loaded, you don't need to name the malicious asset bundle the same thing in this example, as in this PoC uses bettercap to do a MITM and automatically swaps the requested file with the specified malicious one, all it cares about is the prefab name in the upper left "friend", that's what the entry point for the content in the file to spoof actually is.

At this point, I'm going to create the malicious attacking file.

The attacking file, note how the name "friend" is the same as in the legitimate file. This is absolutely critical that the prefab has the same name.

This is the flow of the attacking entity. Note the attacker rotates instead of translates. Again, the visual script could do anything that is included as enabled in the visual scripting system of the app itself, or could even potentially create new functionality such as facial recognition from existing primitives.

Once the malicious entity is set up, just package it up using the AssetBundleBrowser build tool.

Phase 2: Exploitation

Start bettercap and run a MITM. In this case, the data is sent over HTTP, so you need to use the HTTP module for bettercap. Now let's see this in action. Please note that the asset bundles in this PoC are having issues with the materials which is why they are pink, but that can be fixed.

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

Remember that the first one moving forward is legitimate behavior, the second rotating one is Meilicious

That's literally all you have to do to exploit this, and it works in the most recent version of Unity3D.