# See what it's like to have a partner in the fight.

redcanary.com/blog/threat-research-questions

In a recent blog post, we introduced you to AtomicTestHarnesses, one of the ways Red Canary's threat research team iteratively improves detection coverage. In this post, we will highlight the philosophy and methodology that goes into understanding an attack technique, defining its scope, and developing test harness code for the purpose of validating detection pipelines. This process encourages analysts to ask more specific, mindful questions in pursuit of their detection and prevention goals.

## Attack technique research workflow in action

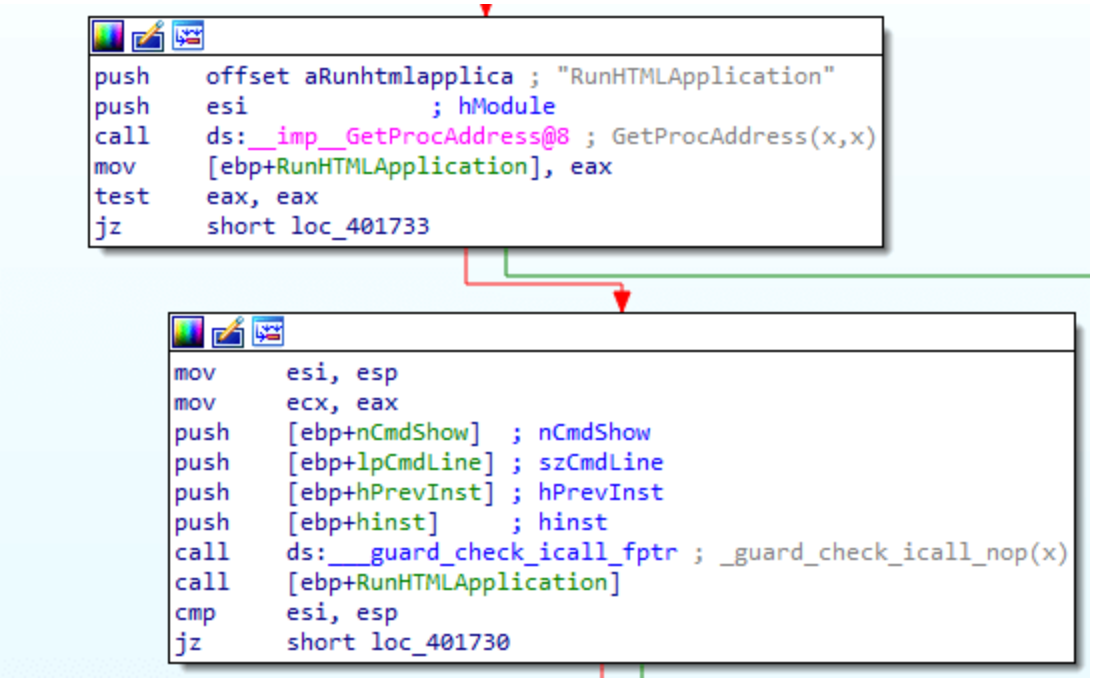
Long before implementing code, you should have a good sense of the scope of the technique at hand, and that's what this research process helps to uncover. Continuing the thread from our last blog post, using MSHTA as our example, we'll want to ask ourselves, "from a detection perspective, **how would we think about detecting suspicious usage of MSHTA?**" That being a very broad question, we need to scope the problem to avoid going down too many research rabbit holes that may ultimately deviate from the technique at hand.

So to start scoping, we ask ourselves the following question: **what exactly, at a technical level, do we define MSHTA to be?**

### Step 1: Define and scope the technique

In order to define what MSHTA is, we start with what we implicitly know it to be and then ask leading questions from there. The easiest way to start with what we know is to look to open source intelligence and identify how attackers abuse MSHTA. For example, we know that attackers execute malicious code with MSHTA using both `mshta.exe` —the supported, built-in utility for doing so—and `rundll32.exe`, which appears ("appears" being an intentionally speculative word requiring clarification) to be a non-standard method of executing HTA content by calling the RunHTMLApplication function within `mshtml.dll`.

So is MSHTA defined by `mshta.exe` and `rundll32.exe` (and nothing else)? Well, not quite, since `rundll32.exe` is a general purpose utility used to execute specifically crafted DLL export functions. The `rundll32.exe` execution, however, can offer a hint as to the core of what makes MSHTA… MSHTA. Some light reversing of `mshta.exe` reveals that the executable is no more than a simple wrapper for the `RunHTMLApplication` function in `mshtml.dll`:




The common component that invokes HTA functionality appears to be the `RunHTMLApplication` function. After arriving to that conclusion, we now have the required vocabulary to ask the following questions to help further refine our scope:

- **Can any other built-in utilities be used to invoke `RunHTMLApplication` functionality?**
- **What advantage, if any, would an attacker have in building their own tool to interface with the `RunHTMLApplication` function?**

## Expand the scope beyond in-the-wild usage

Can any other built-in utilities be used to invoke HTA functionality? The short answer is, no. We performed a sweep of all binaries that might invoke the `RunHTMLApplication` function and found no additional binaries that would yield direct execution of HTA script code. Does that mean that no such binary exists? Of course not. But we were content with the level of due diligence applied to answer the question at the time it was posed. And if anyone discovered another signed HTA host binary that could be easily weaponized, we could very quickly improve our coverage by incorporating that variant into our existing automation of the technique. With threat research and detection engineering, as with any other discipline, we must always remain mindful to not let perfect be the enemy of good. “Perfection” comprises an infinite number of rabbit holes for which there is no fixed destination.

> With threat research and detection engineering, we must always remain mindful to not let perfect be the enemy of good. “Perfection” comprises an infinite number of rabbit holes for which there is no fixed destination.

### Finalize the initial scope

From an evasion perspective, we pondered what advantage an attacker would have in building their own tool to interface with the `RunHTMLApplication` function. As trivial as it would be for an attacker to implement their own code to invoke malicious HTA content, we were unclear on what it would buy them if they already have the means to execute arbitrary code. In other words, **what additional evasion opportunities would it buy an adversary?** We couldn’t come up with a compelling evasion justification.

Now, because we arrived at this conclusion, does that mean that we should not care about attackers directly interfacing with the `RunHTMLApplication` function? Absolutely not. In fact, were an attacker to do such a thing, we might be able to detect such behavior as an anomaly. Ultimately though, as Jeffrey Snover eloquently puts it, “to ship is to choose.”

To decide on what HTA functionality to automate, we needed to define the scope of what would be implemented. After considering our questions and subsequent investigations, we decided to focus automation of HTA script code around only `mshta.exe` and `rundll32.exe`. We are confident in this decision; it buys us a ton of coverage, and we can easily extend our automation to support new variations should they become operationally viable.

## Step 2: Identify technique variations

With the scope defined, now what? This is where the fun begins! Now that we’ve narrowed our scope down to automating HTA script execution via `mshta.exe` and `rundll32.exe`, we can now ask more targeted questions. Specifically, **what inputs does an attacker have control over to influence execution and potentially evade naive detections?** This

question often involves more in-depth research, and in the interest of time, we won't delve into the technical specifics. Through the course of our efforts, we honed in on the following attributes that an attacker had direct control over:

1. The HTA filename can be any name and any file extension that isn't associated with the "text/plain" MIME type (e.g., an extension of .txt will result in displaying but not executing HTA script content).
2. A URI can be specified from where HTA content is first downloaded. It turns out that a URI in Windows terminology is an instance of a protocol handler, a piece of code that is responsible for parsing and interpreting strings that begin with the following format: "handler_name:" (e.g., "https:", "javascript:", "about:", etc.)
3. Different script engines can be supplied in HTA content. We needed to determine what script engines were available and which ones facilitated the execution of arbitrary code. This script engine dictates a specific DLL image load that would occur (e.g., `vbscript.dll`, `jscript9.dll`, `jscript.dll`, etc.).
4. Protocol handlers (e.g., "vbscript", "javascript", "about") can be specified to influence how inline HTA content can be executed, i.e., without needing to drop HTA content to disk. We needed to enumerate the available protocol handlers and then identify which ones led to direct code execution.
5. HTA content can be embedded and executed from within other file formats. Learning of this is also what led to our discovery of CVE-2020-1599.
6. HTA content can be executed remotely via UNC paths.
7. HTA exposes a COM interface that is remotely accessible, making HTA execution a viable option for lateral movement.
8. .hta files have a default file handler, meaning that they can be executed by double clicking on them or invoking them with "explorer.exe foo.hta".
9. An attacker has full control over the path and filename of `mshta.exe` and `rundll32.exe`.

Every single one of the variations in which an attacker realistically has control over inputs to influence HTA script execution with `mshta.exe` and `rundll32.exe` ought to be automated in a way that is sufficiently abstracted to allow non-subject matter experts control over those points of influence. And this is exactly what we implemented in the HTA test harness in AtomicTestHarnesses, `Invoke-ATHHTMLApplication`.

```
NAME
    Invoke-ATHHTMLApplication

SYNOPSIS
    Test runner for HTML Applications (HTA) for the purposes of validating
    detection coverage.

    Technique ID: T1218.005 (Signed Binary Proxy Execution: Mshta)


SYNTAX
    Invoke-ATHHTMLApplication [-HTAFilePath <String>] [-ScriptContent
    <String>] [-ScriptEngine <String>] [-AsLocalUNCPath]
    [-SimulateLateralMovement] [-MSHTAFilePath <String>] [-TestGuid <Guid>]
    [<CommonParameters>]

    Invoke-ATHHTMLApplication [-HTAFilePath <String>] [-ScriptContent
    <String>] [-ScriptEngine <String>] -SimulateUserDoubleClick [-TestGuid
    <Guid>] [<CommonParameters>]

    Invoke-ATHHTMLApplication -HTAUri <String> [-MSHTAFilePath <String>]
    [-TestGuid <Guid>] [<CommonParameters>]

    Invoke-ATHHTMLApplication [-ScriptEngine <String>]
    [-InlineProtocolHandler <String>] -UseRundll32 [-Rundll32FilePath
    <String>] [-TestGuid <Guid>] [<CommonParameters>]

    Invoke-ATHHTMLApplication [-ScriptEngine <String>]
    [-InlineProtocolHandler <String>] [-MSHTAFilePath <String>] [-TestGuid
    <Guid>] [<CommonParameters>]

    Invoke-ATHHTMLApplication -TemplatePE [-AsLocalUNCPath] [-MSHTAFilePath
    <String>] [-TestGuid <Guid>] [<CommonParameters>]
```

Taking stock in what aspects of a technique an attacker has control over, you may get a better sense of two things:

1. Potentially naive detection logic that an attacker could easily evade
2. The variables that an attacker has less or no control over

In an ideal scenario, the most robust detection logic accounts for everything an attacker has little or no control over. Without performing this level of due diligence with technique research, it can be very difficult to comprehend or quantify how a robust detection would take shape. A robust detection has an arbitrarily longer shelf life than one that does not take attacker-controlled inputs into account.

> The most robust detection logic accounts for everything an attacker has little or no control over.

## Step 3: Identify technique “choke points”

Now that we have a clearer sense of the set of inputs an attacker has to make use of an attack technique, let’s talk about outputs. **What is the set of outputs that a technique might generate that we can potentially use to build detections from?** This is another one of those questions that is overly broad and requires a little bit of deliberate scoping. A more specific question that we might consider first is, **what conditions must be satisfied in order to successfully make use of an attack technique?** In the case of MSHTA, within our established scope of `mshta.exe` and `rundll32.exe`, the following conditions must be met:

1. `mshta.exe` or `rundll32.exe` must execute.
2. Command-line arguments are supplied to `mshta.exe` or `rundll32.exe`.
3. `mshtml.dll` must load as a first step in order to execute script content.
4. One of the DLLs associated with script execution will load depending upon the script engine specified in the HTA. This will be either `vbscript.dll` or `jscript.dll` based on our investigation.

Having a clearer sense of the components required enables us to more narrowly focus potential detection logic and to have a better idea of what, if any, options are available to prevent this technique from being abused. After all, if any link of this chain can be severed, the technique fails. We refer to these “minimum viable” components as attack technique “choke points.”

From a detection perspective, now that we know the necessary components, we can start to identify some potential data needs:

1. We’ll need process creation optics that ideally include optics related to the aspects of the technique that an attacker has control over, which include—but are not limited to—the following:
    - **Executable filename**: So that we can identify when `mshta.exe` or `rundll32.exe` run and whether or not the adversary attempts to rename the file.
    - **Executable path:** So that we can identify if `mshta.exe` or `rundll32.exe` are executing from an expected directory or copied to a location in an attempt to evade naive detection logic.
    - **Process command line:** Because we scoped our research to `mshta.exe` and `rundll32.exe` execution, command-line optics are crucial since an attacker must supply their malicious HTA script code via the command line. Are there methods of evading command-line logging? Yes, but that is a separate attack technique that would warrant its own dedicated research and detection initiative. Remember that we must not fall into too many rabbit holes.

2. It could be useful to have insight into processes that load `mshtml.dll`. While the loading of `mshtml.dll` is implied in our current scope of `mshta.exe` and `rundll32.exe` (in the case of `RunHTMLApplication` being executed), this insight would facilitate future threat hunting. For example, how do we know that attackers will only ever use or abuse `mshta.exe` or `rundll32.exe`?
3. Having insight into processes that load related scripting engine components could be useful down the line, but currently would only be used to differentiate VBScript versus JScript execution. However, knowing that these DLLs load does point to the ability of WSH script components to log script content via the AMSI interface.

From a prevention perspective, we have an initial idea of what components could possibly be blocked. For example, **would it be possible to block the execution of `mshta.exe` or `rundll32.exe` within a specific organization?** Aside from outright blocking executables though, we must consider other preventative mechanisms. Through reverse engineering the `RunHTMLApplication` function in `mshtml.dll` (again, the core component required to invoke HTA functionality), we discovered that if Windows Defender Application Control (WDAC) is in enforcement mode, HTA execution is outright banned. This means that even if an attacker discovered another executable that invoked HTA functionality, or if they interfaced with the `RunHTMLApplication` itself, by default, HTA execution would be blocked. That, in our book, is a very robust mitigation.

## Conclusion

In summary, this research methodology offers the following outcomes:

- The research process focuses as much on the constant refinement of scope as it does gaining further understanding of the technique at hand.
- Enumeration of attack technique variations serves to offer clear insight into the aspects of a technique that an attacker has direct control over and, conversely, what they have little-to-no control over. In an ideal scenario, a detection engineer has the opportunity to build the most robust detection (i.e., resilient against evasion) using logic that depends as little as possible on aspects of a technique that an attacker has control over.
- Tactical identification of attack technique “choke points” determines the minimum set of technical components required where, if any one of those links in the chain breaks, weaponization of the technique fails.
- Knowledge of attack technique choke points further refines the scope of research, which further refines scope for detection and prevention.

While we used MSHTA as an illustrative example, this research process can be applied equally to any attack technique. Effective research is built on a foundation of asking specific, deliberate questions in an attempt to reduce a broad objective (e.g., “can we detect technique X?”) into something more achievable, measurable, and resilient against evasion.

## Appendix

Here is a sampling of the ad-hoc scripts we wrote during our research process to answer some of the questions that arose. Please excuse the lack of cleanliness, as these were designed as one-time use scripts to answer very specific questions.

### Identifying built-in utilities that incorporate HTA functionality

The following code was used to identify any other built-in application beyond `mshta.exe` that calls `RunHTMLApplication`:

```
Get-CimInstance -ClassName CIM_DataFile -Filter 'Drive = "C:" AND (Extension = "dll"
OR Extension = "exe")' -Property 'Name' | % { Get-Item $_.Name |Select-String -
Pattern 'RunHTMLApplication' -Encoding ascii }
```

The above one-liner found no other built-in utilities that make native use of the `RunHTMLApplication` function outside of `mshta.exe` and `rundll32.exe`. Had this yielded any candidate executables, a manual review process would have been required to assess feasibility of its use to execute attacker-supplied HTA script code.

### Identifying available WSH script engines

While running a procmon trace, we observed that the presence of an “OLEScript” subkey within the HKEY_CLASSES_ROOT registry hive is an indication that the WSH scripting engine is being used. It’s known that you can specify VBScript and JScript as the scripting language in HTA, but it was unclear if any other scripting engines were supported. We wrote the following PowerShell function to enumerate other possible WSH scripting engines:

```
function Get-OLEScriptingEngine {
    Get-ChildItem -Path 'Registry::HKEY_CLASSES_ROOT\' | % {
        $Key = $_ | Get-Item

        $HasOLEScript = $Key.GetSubKeyNames() | ? { $_ -contains 'OLEScript' }

        if ($HasOLEScript) {
            # Pull the CLSID of the corresponding script engine
            $CLSID = Get-ItemPropertyValue -Path "Registry::$($Key.Name)\CLSID" -
Name '(Default)'

            $EnginePath = Get-ItemPropertyValue -Path
"Registry::HKEY_CLASSES_ROOT\CLSID\$CLSID\InprocServer32" -Name '(Default)'

            [PSCustomObject] @{
                EngineName = $Key.PSChildName
                CLSID = $CLSID
                EnginePath = $EnginePath
            }
        }
    }
}
```

The PowerShell function yielded the following results:

```
EngineName             CLSID                                    EnginePath
----------             -----                                    ----------
ECMAScript             {f414c260-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
ECMAScript Author      {f414c261-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JavaScript             {f414c260-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JavaScript Author      {f414c261-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JavaScript1.1          {f414c260-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JavaScript1.1 Author   {f414c261-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JavaScript1.2          {f414c260-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JavaScript1.2 Author   {f414c261-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JavaScript1.3          {f414c260-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JavaScript1.3 Author   {f414c261-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JScript                {f414c260-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JScript Author         {f414c261-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JScript.Compact        {cc5bbec3-db4a-4bed-828d-08d78ee3e1ed}
C:\Windows\System32\jscript.dll
JScript.Compact Author {f414c261-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
JScript.Encode         {f414c262-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
LiveScript             {f414c260-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
LiveScript Author      {f414c261-6ac0-11cf-b6d1-00aa00bbbb58}
C:\Windows\System32\jscript.dll
VBS                    {B54F3741-5B07-11cf-A4B0-00AA004A55E8}
C:\Windows\System32\vbscript.dll
VBS Author             {B54F3742-5B07-11cf-A4B0-00AA004A55E8}
C:\Windows\System32\vbscript.dll
VBScript               {B54F3741-5B07-11cf-A4B0-00AA004A55E8}
C:\Windows\System32\vbscript.dll
VBScript Author        {B54F3742-5B07-11cf-A4B0-00AA004A55E8}
C:\Windows\System32\vbscript.dll
VBScript.Encode        {B54F3743-5B07-11cf-A4B0-00AA004A55E8}
C:\Windows\System32\vbscript.dll
VBScript.RegExp        {3F4DACA4-160D-11D2-A8E9-00104B365C9F}
C:\Windows\System32\vbscript.dll
XML                    {989D1DC0-B162-11D1-B6EC-D27DDCF9A923}
C:\Windows\System32\msxml3.dll
```

Not all of these scripting engines support the full functionality that VBScript and JScript would, so we didn't prioritize testing them (e.g., XML, VBScript.RegExp). The primary takeaway here is that while script code can be reflected in different ways based on the script engine, detection artifacts will remain the same (i.e., same DLL loads and same structure to the HTA document). Selection of the scripting engine will affect, however, malicious script analysis. For example, a decoder would be required to interpret JScript.Encode and VBScript.Encode content.

### HTTP[S] hosting of HTA content

It took us a while to realize that HTA content could not be hosted on GitHub and downloaded/executed by `mshta.exe` because GitHub uses a "text/plain" MIME type and HTA content will only display and not execute in that case. To address this, we hosted sample HTA content from a static site where we had control over the MIME type and discovered that not only did specifying a MIME type of application/HTA cause the HTA content to execute, but any MIME type that's not text/plain appears to cause the content to execute. This finding is relatively significant because it reveals that one actor was hosting HTA content appended to a .crl file and the MIME type used was "application/x-x509-ca-cert", which blends in quite well over the network. This finding also helped confirm that any file extension containing HTA content can be downloaded and executed.

The following code was used to enumerate built-in MIME types and their corresponding file associations:

```
Get-ChildItem -Path 'Registry::HKEY_CLASSES_ROOT' | Where-Object {
$_.GetValueNames() -contains 'Content Type' } | ForEach-Object { [PSCustomObject] @{
Extension = $_.PSChildName; ContentType = (Get-ItemPropertyValue -Path
"Registry::HKEY_CLASSES_ROOT\$($_.PSChildName)" -Name 'Content Type') } } | Sort-
Object -Property ContentType
```

This code didn't produce any significant findings but it satiated a curiosity around what MIME types are associated with an array of file extensions in Windows by default.

## References

The following resources were indispensable to our understanding of HTA tradecraft and helped reduce the amount of time required to fully contextualize the technique:

Related Articles

Detection and response

## ChromeLoader: a pushy malvertiser

Detection and response

## Intelligence Insights: May 2022

Detection and response

## The Goot cause: Detecting Gootloader and its follow-on activity

Detection and response

## Marshmallows & Kerberoasting

### Subscribe to our blog