


# Categorizing Software with Code Families

---

 [vertex.link/blogs/categorizing-software-with-code-families/](https://vertex.link/blogs/categorizing-software-with-code-families/)



by savage | 2025-01-22

---

When working on a methodology for tracking software, The Vertex Project analysts wanted an approach that would give us greater precision in documenting our findings and asking questions about our data. In [More Than Malware Families](#), we introduced several categories into which we organize software: code families, software suites, and software ecosystems. This blog will focus on the most fundamental of the three, code families, and describe how other analyst teams might approach creating code families to categorize tools.

## What is a Code Family?

---

If we want to be extremely precise when identifying, categorizing, and tracking certain kinds of software, then we might begin with creating code families. A code family is a set of executable code based on what an analyst has determined to be the same or highly similar source code. The files associated with that family may share an entire code base or a subset of key components (functions) that are unique to or strongly representative of the code family.

Code families are intended to be granular and allow for more precise file identification. A tool consisting of multiple files will often map to several code families, with each family corresponding to one of the files making up the overall tool. For example, what the industry commonly refers to as "[PlugX](#)" typically consists of three files: an executable file (often from a legitimate vendor), a side-loading DLL, and shellcode. To Vertex, only the shellcode that implements the backdoor functionality is part of the PlugX code family. Although the shellcode and side-loading DLL work together, they do not share the same or highly similar source code required if they were to be the same code family. We could optionally create another named code family if we wanted to track the different side-loading DLLs, otherwise, we might simply track the executable and side-loading DLL as part of the PlugX ecosystem.

Code families are not inherently malicious - analysts can create code families to identify software in general and track a broader variety of tools. An analyst may create a code family to track samples of Microsoft's [PsExec](#), for example. Tracking tools, as we noted in our previous blog, can help analysts more easily recognize samples as they come across them, as well as provide context and identify tactics, techniques, and procedures associated with activity of interest.

## Why Create Code Families?

---

Analyst teams can create code families to help with categorizing tools and components, developing detection, and identifying changes in tools over time. With code families, research into a tool begins at the code level as analysts determine which key samples of source code within that tool will serve as the anchor for the resulting code family. After selecting those code samples, which we refer to as anchor functions, the analyst can identify the corresponding files containing those functions and mark them as associated with that code family. From there, an analyst might work to document relationships between different tools, creating Software Suites or Software Ecosystems as appropriate.

While creating code families has its benefits, this approach is not for everyone and is not a necessary starting point for tool identification. For some teams, basing tool identification on code similarities may be too granular an approach and inconsistent with their analysis needs.

## Creating a Code Family

---

There are two main approaches to choose from when it comes to creating code families, one of which allows for higher fidelity but requires greater resources. The choice in methodology will largely depend upon a combination of a team's analysis requirements and available resources. Thus teams should choose the methodology that best aligns with their tasking, analytic outputs, and available resources.

### Approach 1: Basing the Code Family Off of Anchor Functions

---

Of the two approaches, the most high fidelity method for creating a code family involves basing it off of one or more anchor functions representing key aspects of the source code. An anchor function is the seed of the code family cluster, similar to how a threat cluster seed is the starting point for a threat cluster. As such, while a code family can have multiple anchor functions, each should be unique to that code family.

Ideally, an anchor function will be representative of or tied to a key capability of the executable source code. However, in many instances it is not the capability itself that is unique but the way in which it is implemented. For example, some backdoors obfuscate or encrypt the names of API calls made to the host operating system (e.g., CreateFileA) to mask their functionality. These strings are decoded or decrypted at runtime. The specific algorithms (functions) used and their implementation may be unique to the backdoor, and could therefore be a good candidate for an anchor function for the backdoor's code family.

A team's approach to identifying anchor functions will depend upon its analysis requirements and resourcing. The most precise method is also the most resource intensive, as it involves relying on a malware reverse engineer to identify anchor functions through symbolic execution. Teams without dedicated reverse engineering support may use tools like Vivisect, which identifies symbolic functions that analysts can use to select anchor functions.

Symbolic execution is a robust method of identification as it targets the logic behind the instructions found in the code, and will therefore persist across changes in bytes. In contrast, comparing instruction byte code with something like YARA is more fragile and poses a greater risk of false negatives. Analysts working with a reverse engineer can generate less fragile signatures for anchor functions by omitting relocations that would make the code position dependent. This would help ensure that the signatures still match against files containing the same instruction sequences, even if they are loaded at a different address in RAM.

## **Approach 2: Generally Identifying the Existence of a Code Family**

---

Another approach would be to infer the existence of a code family among a set of highly similar samples, rather than identifying specific anchor functions to precisely define the code family. Instead of relying on symbolic code analysis, this tactic involves using static and dynamic analysis to identify similarities implying the existence of a shared code family among files. These similarities may include a combination of strings found in a binary, format strings in a URL, and execution behaviors, among others.

Although this approach is more accessible and less resource-intensive than identifying anchor functions, it also poses a higher risk of false positives. YARA rules that rely on strings and other application data are less accurate than those focused on identifying functions, as the latter targets the code itself, rather than the data the code uses.

While an analyst can select a range of similarities as evidence of a code family, some will be higher fidelity than others. An analyst must therefore be cognizant of what shared traits they are noting as a proximation for the code family, as selecting something insufficiently unique can result in false positives.

## **Code Families in Practice: The Carrotstick Backdoor**

---

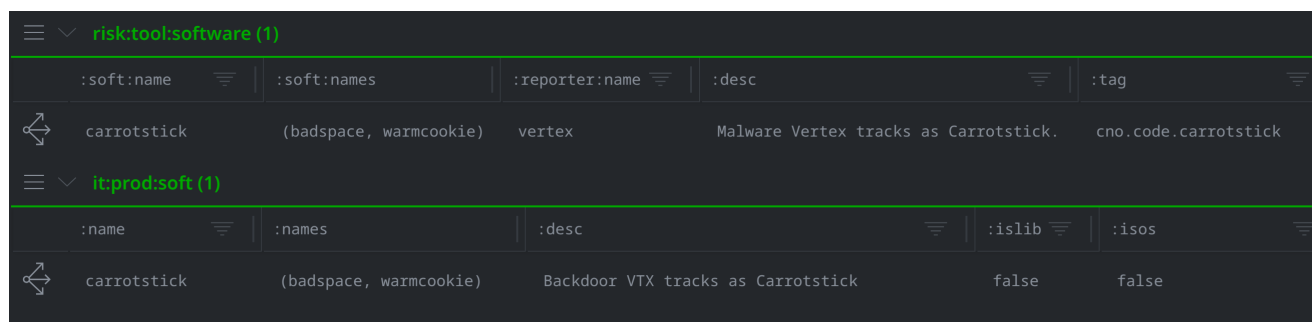
So what does creating a code family look like in practice, and how do we then represent the results in Synapse? Let's take a look at a code family I created for a backdoor Cybersec Sentinel originally reported on in early June 2024. According to CyberSec Sentinel, Elastic, and others, phishing emails use employment-related lures to entice recipients to click on a malicious link, which, after a series of redirects, delivers a Javascript file that downloads a backdoor. While Cybersec Sentinel, Elastic, and others refer to the backdoor as WarmCookie, I opted to name our internal code family Carrotstick to differentiate between our own analysis and that of other organizations.

In this instance, I sought to generally identify the existence of a code family among the backdoor samples, rather than try to specifically identify anchor functions upon which to base the code family. My evidence for the code family included a mix of execution behavior, such as:

- Downloading a DLL to a temp directory with a random name and file extension, while also copying the DLL to `C:/ProgramData/RtlUpd/RtlUpd.dll`;
- Using `rundll32.exe` to launch the DLL with the parameters "`Start, /p`" for persistence;
- Having a hard-coded GUID-like string as a mutex; and
- Communicating with a hardcoded IP address over HTTP.

I also used [Elastic's YARA rule](#), although I edited it to include a different combination of strings and conditions.

After identifying the parameters for the Carrotstick code family, I created a `risk:tool:software` node and linked it to an `it:prod:soft` node to represent it in Synapse:



risk:tool:software (1)				
:soft:name	:soft:names	:reporter:name	:desc	:tag
carrotstick	(badspace, warmcookie)	vertex	Malware Vertex tracks as Carrotstick.	cno.code.carrotstick

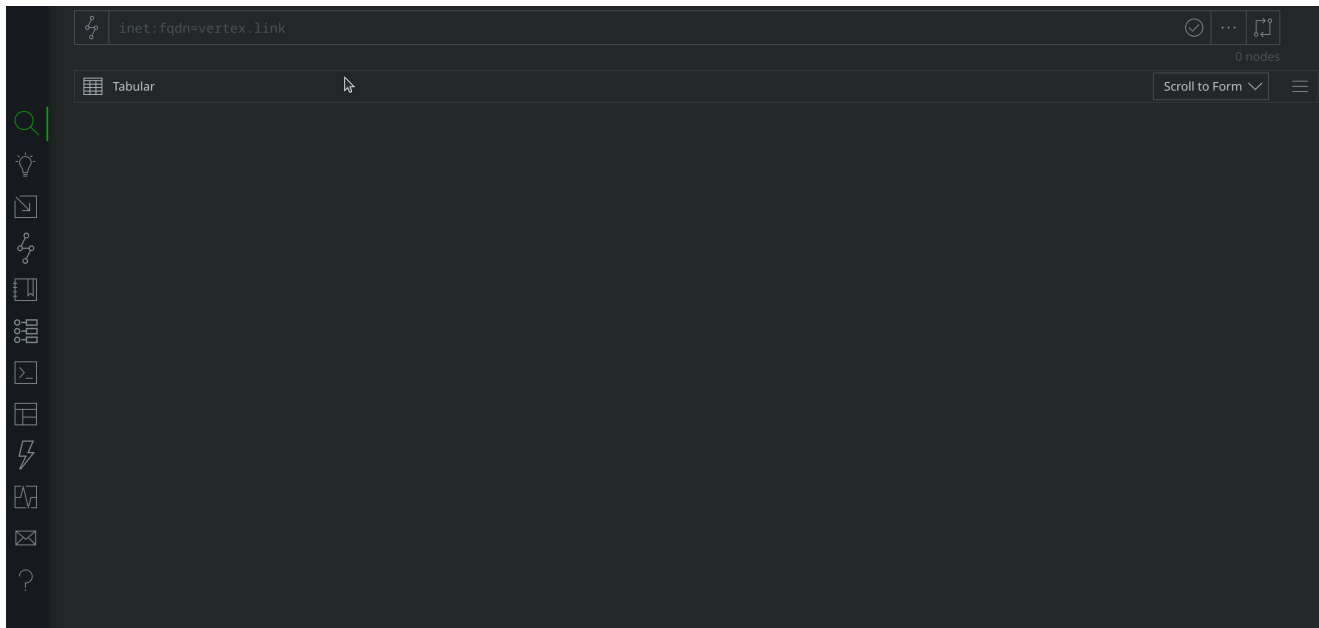
  

it:prod:soft (1)				
:name	:names	:desc	:islib	:isos
carrotstick	(badspace, warmcookie)	Backdoor VTX tracks as Carrotstick	false	false

The `it:prod:soft` node documents Carrotstick as a type of software, while the `risk:tool:software` node (linked to `it:prod:soft` through the `risk:tool:software:soft` property), shows that Carrotstick is a tool associated with malicious activity. I can use additional forms within Synapse to track further details, like versioning information (`it:prod:softver`) and associated techniques (`ou:technique`) as well.

After creating `risk:tool:software` and `it:prod:soft` nodes to represent Carrotstick at a high level, I tagged the `file:bytes` nodes representing Carrotstick samples and their associated `hash:sha256`, `hash:sha1`, and `hash:md5` nodes with `#cno.code.carrotstick` to keep track of them. At this point, I have both higher level details about Carrotstick reflected in Synapse, as well as actual samples of the code family.

You can review this data in the [Vertex Intel-Sharing Instance](#) (register [here](#) for access). In the TLP-Green view, query `risk:tool:software:soft:name=carrotstick` to lift the `risk:tool:software` node representing the Carrotstick backdoor. You can then use the [Explore](#) button to pivot to the associated `syn:tag` nodes (and then again from there, to view all nodes with the `#cno.code.carrotstick` tag), as shown below:



Alternatively, you can query `#cno.code.carrotstick` to lift the tagged nodes directly.

## Categorizing with Code Families

---

Within The Vertex Project, we create code families to allow for greater granularity and precision in identifying software by doing so at the code level. Using code families to categorize software allows us to deconstruct a tool down to the anchor function(s), or key components representing the code family. As noted in this blog, there are multiple approaches that teams may take when it comes to identifying anchor functions and creating code families, from those that offer higher fidelity but are more resource intensive, to those that require fewer resources but a greater risk of false positives. As always, we encourage teams to choose the approach that is most appropriate for their use case.

In a following blog, we'll discuss Software Suites and Software Ecosystems, as well as walk through creating a Software Ecosystem to track indicators associated with our Carrotstick backdoor.