Microsoft open sources CodeQL queries used to hunt for Solorigate activity

February 25, 2021

> **UPDATE:** Microsoft continues to work with partners and customers to expand our knowledge of the threat actor behind the nation-state cyberattacks that compromised the supply chain of SolarWinds and impacted multiple other organizations. Microsoft previously used 'Solorigate' as the primary designation for the actor, but moving forward, we want to place appropriate focus on the actors behind the sophisticated attacks, rather than one of the examples of malware used by the actors. Microsoft Threat Intelligence Center (MSTIC) has named the actor behind the attack against SolarWinds, the SUNBURST backdoor, TEARDROP malware, and related components as NOBELIUM. As we release new content and analysis, we will use NOBELIUM to refer to the actor and the campaign of attacks.

A key aspect of the Solorigate attack is the supply chain compromise that allowed the attacker to modify binaries in SolarWinds' Orion product. These modified binaries were distributed via previously legitimate update channels and allowed the attacker to remotely perform malicious activities, such as credential theft, privilege escalation, and lateral

movement, to steal sensitive information. The incident has reminded organizations to reflect not just on their readiness to respond to sophisticated attacks, but also the resilience of their own codebases.

Microsoft believes in leading with transparency and sharing intelligence with the community for the betterment of security practices and posture across the industry as a whole. In this blog, we'll share our journey in reviewing our codebases, highlighting one specific technique: the use of CodeQL queries to analyze our source code at scale and rule out the presence of the code-level indicators of compromise (IoCs) and coding patterns associated with Solorigate. We are open sourcing the CodeQL queries that we used in this investigation so that other organizations may perform a similar analysis. Note that the queries we cover in this blog simply serve to home in on source code that shares similarities with the source in the Solorigate implant, either in the syntactic elements (names, literals, etc.) or in functionality. Both can occur coincidentally in benign code, so all findings will need review to determine if they are actionable. Additionally, there is no guarantee that the malicious actor is constrained to the same functionality or coding style in other operations, so these queries may not detect other implants that deviate significantly from the tactics seen in the Solorigate implant. These should be considered as just a part in a mosaic of techniques to audit for compromise.

Microsoft has long had integrity controls in place to verify that the final compiled binaries distributed to our servers and to our customers have not been maliciously modified at any point in the development and release cycle. For example, we verify that the source file hashes generated by the compiler match the original source files. Still, at Microsoft, we live by the "assume breach" philosophy, which tells us that regardless of how diligent and expansive our security practices are, potential adversaries can be equally as clever and resourced. As part of the Solorigate investigation, we used both automated and manual techniques to validate the integrity of our source code, build environments, and production binaries and environments.

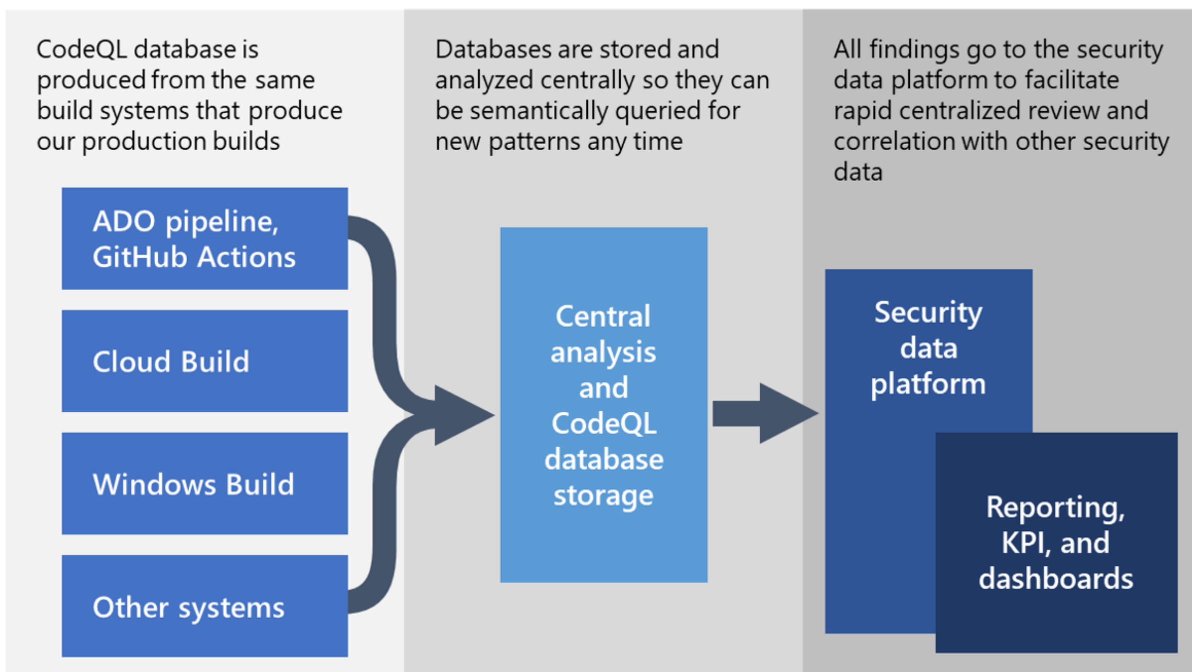
Microsoft's contribution during Solorigate investigations reflects our commitment to a community-based sharing vision described in Githubification of InfoSec. In keeping with our vision to grow defender knowledge and speed community response to sophisticated threats, Microsoft teams have openly and transparently shared indicators of compromise, detailed attack analysis and MITRE ATT&CK techniques, advanced hunting queries, incident response guidance, and risk assessment workbooks during this incident. Microsoft encourages other security organizations that share the "Githubification" vision to open source their own threat knowledge and defender techniques to accelerate defender insight and analysis. As we have shared before, we have compiled a comprehensive resource for technical details of the attack, indicators of compromise, and product guidance at https://aka.ms/solorigate. As part of Microsoft's sweeping investigation into Solorigate, we reviewed our own environment. As we previously shared, these investigations found activity

with a small number of internal accounts, and some accounts had been used to view source code, but we found no evidence of any modification to source code, build infrastructure, compiled binaries, or production environments.

A primer on CodeQL and how Microsoft utilizes it

CodeQL is a powerful semantic code analysis engine that is now part of GitHub. Unlike many analysis solutions, it works in two distinct stages. First, as part of the compilation of source code into binaries, CodeQL builds a database that captures the model of the compiling code. For interpreted languages, it parses the source and builds its own abstract syntax tree model, as there is no compiler. Second, once constructed, this database can be queried repeatedly like any other database. The CodeQL language is purpose-built to enable the easy selection of complex code conditions from the database.

One of the reasons we find so much utility from CodeQL at Microsoft is specifically because this two-stage approach unlocks many useful scenarios, including being able to use static analysis not just for proactive Secure Development Lifecycle analysis but also for reactive code inspection across the enterprise. We aggregate the CodeQL databases produced by the various build systems or pipelines across Microsoft to a centralized infrastructure where we have the capability to query across the breadth of CodeQL databases at once. Aggregating CodeQL databases allows us to search semantically across our multitude of codebases and look for code conditions that may span between multiple assemblies, libraries, or modules based on the specific code that was part of a build. We built this capability to analyze thousands of repositories for newly described variants of vulnerabilities within hours of the variant being described, but it also allowed us to do a first-pass investigation for Solorigate implant patterns similarly, quickly.



We are open sourcing several of the C# queries that assess for these code-level IoCs, and they can currently be found in the CodeQL GitHub repository. The Solorigate-Readme.md within that repo contains detailed descriptions of each query and what code-level IoCs each one is attempting to find. It also contains guidance for other query authors on making adjustments to those queries or authoring queries that take a different tactic in finding the patterns.

GitHub will shortly publish guidance on how they are deploying these queries for existing CodeQL customers. As a reminder, CodeQL is free for open-source projects hosted by GitHub.

Our approach to finding code-level IoCs with CodeQL queries

We used two different tactics when looking for code-level Solorigate IoCs. One approach looks for particular syntax that stood out in the Solorigate code-level IoCs; the other approach looks for overall semantic patterns for the techniques present in the code-level IoCs.

The syntactic queries are very quick to write and execute while offering several advantages over comparable regular expression searches; however, they are brittle to the malicious actor changing the names and literals they use. The semantic patterns look for the overall techniques used in the implant, such as hashing process names, time delays before contacting the C2 servers, etc. These are durable to substantial variation, but they are more complicated to author and more compute-intensive when analyzing many codebases at once.

Sample technique from the implant

```
//The Implant utilizes the below code to hash the name of running processes (the s parameter passed in)
//using the FNV-1A algorithm.  It them XOR's the resulting hash with the value
//6605813339339102567UL.  This technique allows them to both obfuscate the process names they are looking
//for as derived hashes AND use different derived hashes in different campaigns by changing the XOR value.
//Its a clever technique to evade most traditional malware string analysis BUT it gives a very distinct
//technique to run a semantic query to detect
11 references
private static ulong GetHash(string s)
{
    //the implementation of FNV-1A
    ulong num = 14695981039346656037UL; /* NOT A HASH - FNV base offset */
    try
    {
        foreach (byte b in Encoding.UTF8.GetBytes(s))
        {
            num ^= (ulong)b;
            num *= 1099511628211UL; /* NOT A HASH - FNV prime */
        }
    }
    catch //another common technique in the implant was to swallow all exceptions
    {
    }

    //the XOR with a campaign specific value to generate different derived
    //hashes for different campaigns
    return num ^ 6605813339339102567UL;
}
```

CodeQL query to find the technique

```
from Variable v, Literal l, LoopStmt loop, Expr additional_xor
where
  maybeUsedInFNVFunction(v, _, _, loop) and
  (
    exists(BitwiseXorExpr xor2 | xor2.getAnOperand() = l and additional_xor = xor2 |
      loop.getAControlFlowExitNode().getASuccessor*() = xor2.getAControlFlowNode() and
      xor2.getAnOperand() = v.getAnAccess()
    )
    or
    exists(AssignXorExpr xor2 | xor2.getAnOperand() = l and additional_xor = xor2 |
      loop.getAControlFlowExitNode().getASuccessor*() = xor2.getAControlFlowNode() and
      xor2.getAnOperand() = v.getAnAccess()
    )
  )
select l,
  "The variable $@ seems to be used as part of a FNV-like hash calculation,"
      "that is modified by an additional $@ expression using literal $@.",
  v, v.toString(), additional_xor, "xor", l, l.toString()
```

By combining these two approaches, the queries are able to detect scenarios where the malicious actor changed techniques but used similar syntax, or changed syntax but employed similar techniques. Because it’s possible that the malicious actor could change both syntax and techniques, CodeQL was but one part of our larger investigative effort.

Next steps with CodeQL

The queries we shared in this blog and described in Solorigate-Readme.md target patterns specifically associated with the Solorigate code-level IoCs, but CodeQL also provides many other options to query for backdoor functionality and detection-evasion techniques.

These queries were relatively quick to author, and we were able to hunt for patterns much more accurately across our CodeQL databases and with far less effort to manually review the findings, compared to using text searches of source code. CodeQL is a powerful developer tool, and our hope is that this post inspires organizations to explore how it can be used to improve reactive security response and act as a compromise detection tool.

In future blog posts, we'll share more ways that Microsoft uses CodeQL. We'll also continue open-sourcing queries and utilities that build upon CodeQL so that others may benefit from them and further build upon them.