Naval Information Warfare Center PACIFIC

TECHNICAL DOCUMENT 3429
FEBRUARY 2024

Hyperledger Enhanced Layered and Integrated Cyber Security (HELICS)

James H. Allphin
Anthony M. Quintero-Quiroga
David Kwon
Barry D. Dudley
NIWC Pacific

.

Naval Information Warfare Center (NIWC) Pacific
San Diego, CA 92152-5001

This page is intentionally blank.

TECHNICAL DOCUMENT 3429
FEBRUARY 2024

Hyperledger Enhanced Layered and Integrated Cyber Security (HELICS)

James H. Allphin
Anthony M. Quintero-Quiroga
David Kwon
Barry D. Dudley
NIWC Pacific

Administrative Notes:

This DOCUMENT was approved through the Release of Scientific and Technical Information (RSTI) process in October 2023 and formally published in the Defense Technical Information Center (DTIC) in FEBRUARY 2024.

NIWC Pacific
San Diego, CA 92152-5001

NIWC Pacific
San Diego, California 92152-5001

P.M. McKenna, CAPT, USN
Commanding Officer

M.J. McMillan
Executive Director

ADMINISTRATIVE INFORMATION

The work described in this report was performed by the 584 Division of the Cyber Science and Technology Department, Naval Information Warfare Center (NIWC) Pacific, San Diego, CA. The project was funded by Project Overhead funds through the Cyber Engineering and Integration Division.

Released by
Bruce Waldron, Division Head
(Cyber Engineering and Integration)

Under authority of
Carly A. Jackson, Department Head
(Cyber Science and Technology)

ACKNOWLEDGMENTS

We would like to acknowledge Bruce Heath, Branch Head of Code 58440 Cybersecurity and HLS Engineering, for his contribution and support of the work contained in this publication. Mr. Heath's drive and will to support his colleagues to achieve greatness will always be remembered.

Editor: RJP

ACRONYMS

ATO	Authorization to Operate
CA	Certificate Authority
CIO	Chief Information Officer
CISA	Cybersecurity and Infrastructure Security Agency
CLI	Command Line Interface
CPU	Central Processing Unit
CUI	Controlled Unclassified Information
DDIL	Denied, Disrupted, Intermittent, and Limited
DLT	Distributed Ledger Technology
DoD	Department of Defense
DoDI	DoD Instruction
DoDIN	DoD Information Network
GAO	Government Accountability Office
HELICS	Hyperledger Enhanced Layered and Integrated Cybersecurity
ISCM	Information Security Continuous Monitoring
NIST	National Institute of Standards and Technology
NIWC PAC	Naval Information Warfare Center Pacific
PoS	Proof-of-Stake
PoW	Proof-of-Work
RMF	Risk Management Framework
SCC	SCAP Compliance Checker
SMS	Short Messaging Service
STIG	Security Technical Implementation Guide
V-ID	Vulnerability Identifier

This page is intentionally blank.

CONTENTS

APPENDIX

FIGURES

1. INTRODUCTION

1.1 PURPOSE

Cyber resiliency is a prevalent topic across the Department of Defense (DoD). New technologies are introduced to the DoD Information Network (DoDIN) almost daily, each time with increased pressure for efficiency. It is a major point of emphasis among senior leaders to increase the rate at which systems can receive an Authorization to Operate (ATO) –while maintaining appropriate vigor in reviewing the security posture of these systems. With an over emphasis on program budgets or timelines, there is an increased risk of implementing new technology which hasn't been adequately secured. This is likely to result in a strain on the entire authorization process. In order to meet the growing need for cyber resiliency in a climate ripe with resource strains and looming deadlines, modern and sophisticated solutions need to be evaluated.

With the most stringent manual scanning and patch management practices in place, systems can still be vulnerable to zero-day attacks, supply chain shortfalls, and the most prevalent attack vector of all – the insider threat. The vulnerability management process itself is only as reliable as the people who are responsible to implement it. The objective of this use case is to test an innovative new solution for continuous monitoring and vulnerability management, which will create an automated, near real-time detection and response to changes in a system's secure baseline configuration.

1.1.1 Objectives

The intention for this use case was to utilize a hybrid approach to system monitoring for better cybersecurity, hereby referred to as Hyperledger Enhanced Layered and Integrated Cybersecurity (HELICS). The targeted outcome was to create a unique identifier for an already secured information system using system-level hashing. The system's hash value would serve as a pseudo-fingerprint to ensure the integrity of the system's secure baseline. The hash value would then be monitored for modifications using blockchain technology. This technology is similar to law-enforcements use of human fingerprints to determine if a crime has been committed, using a hash value of the system's secure baseline adds irrefutable integrity to the system monitoring process. Additionally, the goal was to also demonstrate that blockchain technology offers an automation solution with the use of smart contracts; if the system's secure baseline were compromised or modified without authorization, response activities would be configured to initiate automatically. If successfully implemented, this will create a defense-in-depth solution for continuous monitoring (with irrefutable integrity) and add automation to the vulnerability management process. In fact, the potential benefits of this use case reach beyond the continuous monitoring and vulnerability management process (more on this later).

There are several primary technologies discussed for this use case which are important to have a basic understanding of in order to realize its full benefits:

- STIG Analysis Tools– SCAP Compliance Checker (SCC), CyberKnight, evaluateSTIG, and even ACAS are cybersecurity testing tools used to scan systems for compliance with Security Technical Implementation Guides (STIGs). CyberKnight scans a systems settings, registry, and baseline configuration to determine the status of countless STIG checks and generates a report of the system's overall STIG implementation status. The STIG data outputs to a report which is referred to as a STIG "Checklist," which contains several data fields: Vulnerability Identifications (V-IDs) and two data fields where the tester can provide detailed information – "Comments" and "Finding Details." The tests for this use case monitor the V-IDs and Finding Details data fields in the STIG Checklist in order to

identify secure baseline changes. The following are STIG evaluation tools which have been tested with HELICS:

  - CyberKnight – CyberKnight is a STIG analysis tool developed by Naval Information Warfare Center (NIWC) Pacific Code 58440. CyberKnight is a lightweight (16MB) executable that can scan Linux based systems for applicable STIGs and generate a single STIG checklist consisting of the OPEN and CLOSED findings from the scan. The FINDING DETAILS section of the checklist is populated with useful information that can be used to remediate OPEN findings. Currently, CyberKnight automates about 90% of all applicable STIGs.
  - SCAP Compliance Checker (SCC) – SCC is the *defacto* standard in STIG Compliance auditing and is also supported by HELICS.

- Blockchain Technology – Blockchain Technology is a chain-of-custody system in which a record of transactions is maintained across several computers that are linked via a peer-to-peer network. The technology was first utilized as a ledger for transactions for various cryptocurrencies, but the benefits of this decentralized managed ledger have greater reaching impacts. Non-repudiation and reliability are critical components to a proper chain-of-custody in cybersecurity, especially for incidents involving cybercrime or espionage. Blockchain Technology offers unparalleled non-repudiation and reliability (Centieiro, 2021). The following Blockchain components were utilized for HELICS:
  - Hyperledger Fabric – Hyperledger Fabric is an open-source enterprise-grade distributed ledger technology (DLT) platform by the Linux Foundation which is compatible with a number of consensus blockchain algorithms (Hyperledger Foundation, 2023).
  - RAFT Consensus Mechanism – The consensus mechanism of a blockchain is the steps taken by the blockchain and its peers to achieve confirmation on the ledger's current state. (For example, if a bank utilized blockchain peers to coincide with different branch locations, the consensus mechanism would be the steps taken by the bank's branches to ensure that each ends up with the same bank ledger after each transaction committed). Popular blockchains, such as Bitcoin and Ethereum, are notorious for using a Proof-of-Work (PoW) consensus mechanism that is resource intensive. Hyperledger Fabric uses a Proof-of-Stake (PoS) consensus mechanism which is a low-resource alternative to PoW consensus. PoW consensus requires expensive and highly specialized computers to run, while PoS blockchains can run on any computer. Hyperledger Fabric's PoS consensus is also widely compatible and easily scalable and better serves to meet user demand. The RAFT Consensus mechanism is utilized by Hyperledger Fabric in order to ensure correct ordering of data on the ledger. It is imperative that communications over a distributed network are properly synchronized so ack responses are presented in the same order which they are sent. Much like a chat between two recipients, if the communications are not properly synchronized, improper order of the correspondences could be detrimental (Hyperledger Fabric, 2023).
  - Smart Contracts – Smart Contracts (i.e., "chaincode") are preconfigured code which are stored on a blockchain. Smart contracts are passive until certain conditions, or criteria, are met. When the pre-programmed conditions are met, smart contracts will trigger the execution of automated events. Smart Contracts work like "if-then-else" statements, they can be written to execute one action if

certain criteria occur, and even trigger a separate workflow immediately afterward. Smart contracts are also recorded on the ledger and a historical record of smart contracts executed can be viewed by permissioned entities. The autonomous nature of smart contracts is beneficial to building efficiency into the continuous monitoring and vulnerability management process. Smart contracts in Hyperledger Fabric support multiple programming languages such as Go, Java, and JavaScript (IBM, 2023).

- HELICS Agent – The HELICS Agent is a small agent which conducts the STIG analysis and attempts to write to and monitor the Hyperledger. It is programmed using Go, and its sole purpose is to automatically perform STIG Analysis using a supported tool (i.e., CyberKnight or SCC), push generated baseline hashes to the Hyperledger and monitor the Hyperledger for relevant changes.

1.1.1.1 Create a secure system baseline using STIG Analysis Tools

The first objective for this use case was to create a secure system baseline to monitor for changes. This would require two separate actions. The first action would be to implement any applicable Operating System and application STIGs in order to create a secure system baseline. The second action would be to scan the system with a STIG Analysis tool to capture the state of the secure baseline. This would establish the baseline for the system which would be monitored for unauthorized changes using the following methods.

1.1.1.2 Create a hash value of the secure system baseline

Creating a secure system baseline using STIGs is an already widely accepted practice throughout the DoD. Once a system receives its Authority-to-Operate (ATO) and deploys to the DoDIN however, program offices with limited time and resources often fall behind schedule reviewing STIG compliance. Monthly scans are conducted, but apart from this, STIGs are oftentimes only reviewed quarterly. This leaves a gap in system monitoring, creating a potential attack vector to the DoDIN. For those reasons, this use case took into consideration a solution for improving system monitoring. The objective was to leverage a practice often used in after-action events during cyber forensic and analysis – creating a hash value of the system.

For this use case, the goal was to generate a unique hash value for each individual STIG check V-ID examined by the STIG Analysis tool. Each of those hash values would then be hashed again to generate a single hash value which would represent the overall secure system baseline *(which is referred to moving forward as the system's "unique identifier")*. Like STIG'ing, using hash values for cybersecurity is already a commonly adopted practice to ensure the integrity of a system's data throughout a chain-of-custody following a security incident – the objective for this use case was to leverage this practice prior to deploying a system to the DoDIN. This would allow for better continuous monitoring; essentially, if the hash value for the secure system baseline were to change at any point in time, it should be assumed that the system was no longer secure and had been compromised or its configuration was changed without authorization.

1.1.1.3 Configure the blockchain ledger to monitor the secure system baseline

Blockchain Technology is a more adequate solution for continuously monitoring a system's secure baseline, in this case via a hash value. Once the system's secure baseline was captured via periodical STIG analys using an installed agent and uniquely identified via a hash value, the hash value would then be submitted by the HELICS agent to the blockchain ledger (Hyperledger Fabric) for continuous monitoring. This is where the benefits of this use case would bear fruit. Not only would the hash

value of the secure baseline create a more accurate representation of the system for continuous monitoring, but using Hyperledger Fabric to monitor the system's hash for changes would add greater integrity to the continuous monitoring process as well as allow for automation in the vulnerability management process. This is a true defense-in-depth approach.

For an adversary to compromise this secure system baseline, he/she would need to not only compromise the system itself, but the blockchain ledger as well. If the system itself were to become compromised, it would be detected on the next periodical STIG analysis in which the installed agent attempts to submit a different hash value (resulting from a modified system due to adversarial action or unauthorized configuration changes.) This is superior to any known mitigation against zero-day attacks as well as vulnerabilities in supply-chain problems.

1.1.1.4 Create an automated vulnerability management action using smart contracts

Once the system's secure baseline is established and non-repudiation is achieved using system hashing, the next objective for this use case was to create automation using smart contracts. As previously described, smart contracts can be configured within Hyperledger Fabric to trigger the execution of automated events when certain conditions or criteria are met. For this use case, the goal was to configure the blockchain ledger to monitor the system's secure baseline hash value for modifications and trigger some sort of response activity immediately upon discovering any changes to the hash value.

1.2 BACKGROUND

To understand the goal and potential benefits for this use case, there are a few key topics to understand. First, this solution seeks to improve the continuous monitoring and vulnerability management process for the DoD, so it is important to understand the guidelines for each process. Gaining better insight into how continuous monitoring and vulnerability management are handled by the DoD offers the opportunity to see where improvements can be made. The objectives for the use case were evaluated for additional insight – with research into different blockchain methods and the feasibility of Smart Contracts for automation. This section of the paper outlines some of the information evaluated for this use case.

1.2.1 Literature Review

The DoD currently uses the NIST SP 800-137, Information Security Continuous Monitoring (ISCM) for Federal Information Systems as its guide for continuous monitoring. This NIST guide outlines a six-step approach to continuous monitoring. The process begins with defining the Information System Continuous Monitoring (ISCM) Strategy. Once leadership sets the vision for ISCM, it must establish written policy, pass down the guidance and implement the ISCM strategy. For the DoD the ISCM must directly align with the organization's risk tolerance and be adaptive to organization changes and structure. Figure 1 shows the Information System Continuous Monitoring (ISCM) process.

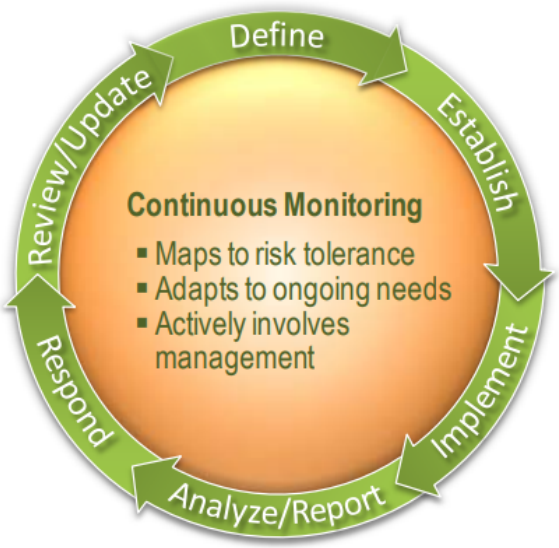


Figure 1. Information System Continuous Monitoring (ISCM) process, (NIST SP800-137).

This use case was less concerned with the DoD's ISCM process, and more concerned with how successfully the ISCM process has been implemented within the DoD. The NIST guide has an entire section dedicated to the "Role of Automation in ISCM" (Section 2.3). This section of the NIST guide opens with the following: "When possible, organizations look for automated solutions to lower costs, enhance efficiency, and improve the reliability of monitoring security-related information. Security is implemented through a combination of people, processes, and technology. The automation of information security deals primarily with automating aspects of security that require little human interaction. Automated tools are often able to recognize patterns and relationships that may escape the notice of human analysts, especially when the analysis is performed on large volumes of data" (Dempsey, et al., 2011)

The DoD's ISCM process lends itself to the use of automation, but its vulnerability management process requires a great deal of manual process and human interaction. Within the DoD, program offices are often given leeway to develop internal processes to manage vulnerabilities and system patching. The process typically consists of system scanning, log review, and manual system patching. If vulnerabilities are discovered through scans, the program will grade the level of risk (typically following industry guidelines) and choose to either mitigate, accept, or remediate the risk. This is a very general description of how vulnerabilities are handled.

This document will not go into more detail to explain the DoD's continuous monitoring or vulnerability management process, but instead seek to explore the overall efficacy of how each process is managed. While the DoD itself does not publish much data regarding how effective its vulnerability management process is, there are cybersecurity providers in the private sector that can be used as a basis of how well industry handles vulnerabilities in continuous monitoring. One such company, Mandiant, releases an annual report detailing trends in the cybersecurity industry. The report, FireEye Mandiant Services Special Report (M-Trends) tells a grim story. For example, the average median dwell time for 2020 was 24 days. "Dwell time is calculated as the number of days an attacker is present in a victim environment before they are detected" (FireEye Mandiant, 2021). In this report, 24 days of dwell time is presented as a massive win for the cyber community as this number has been drastically reduced from years past. For an organization tasked to protect data

which is vital to the warfighter however, 24 days could be the difference between life and death. Without perspective, no Chief Information Officer (CIO) would want to hear that his/her organization allowed a cyber threat to dwell undetected for almost a month.

If anything is to be gleaned from prior cyberattacks, reducing the time to detection is one of the most vital missions for producing a more cyber-resilient organization. The SolarWinds attack is an example of how important this is. The U.S. Government Accountability Office (GAO) released a report on the SolarWinds attack, stating "The cybersecurity breach of SolarWinds' software is one of the most widespread and sophisticated hacking campaigns ever conducted against the federal government and private sector" (U.S. Government Accountability Office (GAO), 2021). The attack was carried out by a Russian Foreign Intelligence Service by injecting a snippet of Trojan code into a software update for SolarWinds' suite of network management and monitoring applications called Orion. Once the software update was installed, it installed a hidden backdoor program on the infected computer which allowed the perpetrator to remotely exploit the infected network.

> In response to this breach, on December 13, 2020, the Department of Homeland Security's Cybersecurity and Infrastructure Security Agency (CISA) released an emergency directive outlining required mitigations for federal agencies to prevent further exploitation of federal information systems. On December 16, the White House's National Security Council activated the Cyber Unified Coordination Group, who is responsible for coordinating the government-wide response to the incident. This group includes officials from the Office of the Director of National Intelligence, FBI, and CISA, with support from the National Security Agency (U.S. Government Accountability Office (GAO), 2021).

The malicious code was first injected into the Orion platform in February 2020. SolarWinds began distributing the software to its customers a month later, and the infiltration went undetected until November 2020 when a cybersecurity professional service firm notified SolarWinds of the breach. SolarWinds estimated that 18,000 customers received a copy of this software update. It went undetected for over 8 months. The SolarWinds attack is the poster child of why it is imperative to implement new and innovative ways for early threat detection – the GAO report states: "Although our examination of SolarWinds is ongoing, we have previously reported on IT supply chain risks and major cybersecurity challenges. We continue to emphasize that the federal government needs to move with greater urgency to improve the nation's cybersecurity as the country faces grave and rapidly evolving threats. Ensuring the cybersecurity of the nation has been on our High-Risk List since 1997" (U.S. Government Accountability Office (GAO), 2021). This paper seeks to present a solution for near-immediate detection of threats like the one involved in the SolarWinds attack.

2. METHODS AND MATERIALS

2.1 METHODS OVERVIEW

This section contains is a procedure list for the HELICS proof-of-concept. This serves to outline the steps which were taken in order to develop and configure the final product.

2.2 METHODS

2.2.1 Create a secure system baseline using STIG analysis tools

- Securely configure a target system utilizing authorized DISA STIGs to a condition where the system is considered securely configured and would be considered production ready, authorized and secure.
- Perform STIG analysis using CyberKnight, SCC, or evaluate STIG to generate a DISA STIG checklist.

2.2.2 Create a hash value of the secure system baseline

- Use a one-way hashing algorithm to hash the contents of each STIG's Vulnerability ID's (V-IDs) and Finding Details data fields, resulting in something similar to:
  - v-111111 <hash of finding details>
  - v-111112 <hash of finding details>
  - v-111113 <hash of finding details>
- Process all V-IDs excluding automated checks which affect the state of the systems hard drive (as these are dynamic and subject to change at intervals which cannot be adequately tracked).
- Using a one-way hashing algorithm, hash all the resulting V-ID's and Finding Details hashes into a single "authorized baseline" hash, which reflects the sum total of the systems secure baseline.

2.2.3 Configure Hyperledger Fabric to monitor the secure system baseline

- Submit this hash to the Hyperledger.
- The authorized baseline hash is written to the Hyperledger.
- Perform STIG analysis to a set periodicity (i.e., every five minutes). Perform the above methods again.
- Attempt to submit new hash to the Hyperledger. If system configuration has changed, the "baseline hash" will be different, reflecting that some configuration has changed on the system.

2.2.4 Create an automated vulnerability management action via smart contracts

- The Hyperledger checks the previous "authorized" hash, and if the new hash does not match the old one, a smart contract is initiated to take some sort of action. If the hash does match the old one, no action is taken, and the hash is written to the ledger.

- The new “unauthorized” hash is written to the Hyperledger to maintain a record.
- Repeat for all systems under monitoring.

2.2.5 Study Design

Hyperledger Fabric was set up by using a default test network. The default test network consists of four Docker containers: two peered containers hosting test peers (named “Org1 peer0” and “Org2 peer0” to represent two unique organizations), a third “Orderer Node” container which is responsible for maintaining the order of the ledger’s transactions, and a final container which is responsible for handling a set of pre-configured tools and CLI commands (only accessible to privileged users). These components combine to form a channel configuration (CC). The channel configuration block contains a list of all organizations which are able to join and make transactions on the channel.

With Hyperledger Fabric, each organization is able to have its own administrator as well as its own Certificate Authority (CA). For smart contract, Hyperledger Fabric allows different endorsement policies and transactional approvals which can be configured for different organizations. Each peer node in an organization is responsible for its own submissions and read/write access to the Hyperledger. When a node receives a transaction from an asset, the node validates the transaction with each of its peers on the blockchain. Before the transaction is committed it to the ledger, an orderer node properly and efficiently orders the transaction submitted by peers in each organization so the ledger can maintain a proper record of all transactions. A diagram of the Hyperledger Fabric Test Network Configuration is shown in Figure 2.

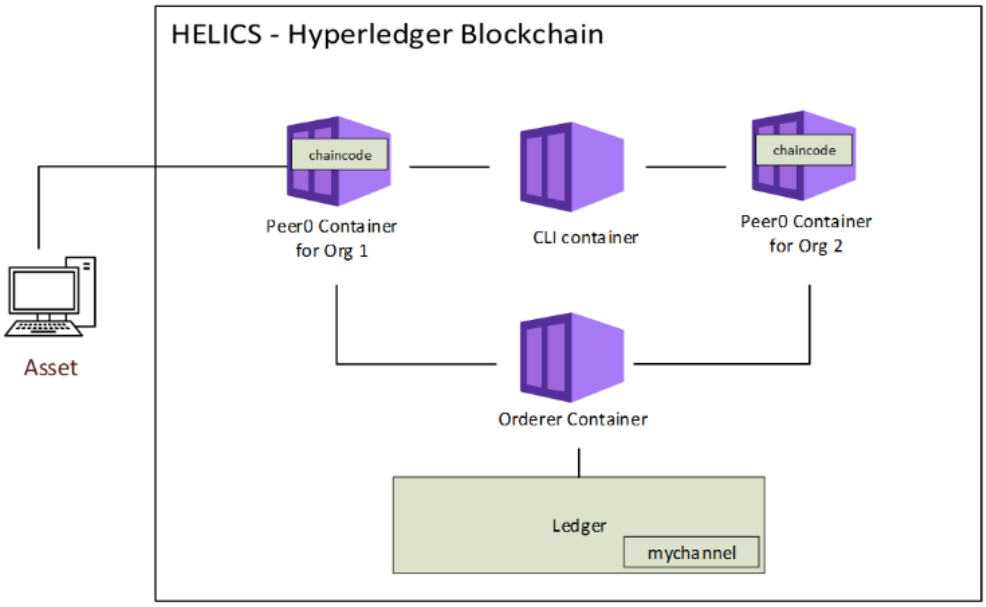


Figure 2. Hyperledger Fabric Test Network Configuration.

This study used peer0 on organization 1 to submit blocks to the Hyperledger.

The target asset which was monitored by STIG analysis was the host system of the docker containers running the Fabric test-network.

The target asset was analyzed using the Ubuntu 20.04 Operating System DISA STIG and two STIG analysis tools were used – CyberKnight and the SCAP Compliance Checker.

The periodicity of the STIG analysis was set to every five minutes.

The HELICS agent was written using the programming language Go which performs the STIG analysis on the target asset every five minutes, generates the hashes, and submits each hash to the Hyperledger using *peer0.org1.example.com* as seen in Figure 3:

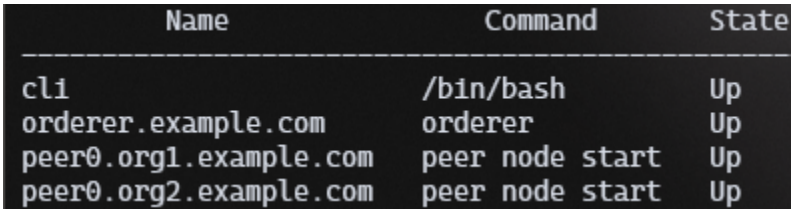


Figure 3. Hyperledger docker based reference test-network.

Configuring Hyperledger and issuing commands is traditionally done via command line interface. However, this use case utilized custom bash scripts to interact with the Hyperledger. Scripts were created which can initiate the Hyperledger, edit the Hyperledger and read from the Hyperledger. The scripts allow for automation and repeatability.

List of scripts used for Hyperledger interactions:

- Asset-exists.sh
    - To check if an existing asset exists on the Hyperledger. Returns true or false based on the query. Ensures queries are functional.
- Compare-hash.sh
    - To compare a test hash against the currently installed hash on the Hyperledger. Returns true or false based on the comparison. Allows for testing validation logic.
- Create-asset.sh
    - Manually creates a new asset to continuously monitor via the Hyperledger.
- Delete-asset.sh
    - Manually deletes an asset on the Hyperledger.
- Get-all-assets.sh
    - Returns all assets currently on the Hyperledger.
- Get-history.sh
    - Returns all history of an asset on the Hyperledger (with timestamps). Useful for tracing down how & when different hashes were submitted to the Hyperledger.
- Read-asset.sh
    - Returns query data for one asset currently on the Hyperledger.
- InstallCC.sh

    - Automatic script to install the “chain-code” or smart contract capability on the Hyperledger.

- SetOrgEnv.sh
    - Sets environment variables for easier transaction and querying of the data on the Hyperledger.
- Update-asset.sh
    - Manually updates an asset on the Hyperledger.
- Update-hash.sh
    - Manually updates only the hash value of an asset on the Hyperledger.

Custom “chaincode” or “smart contracts” were written into the Hyperledger in order to test for vulnerability management automation. The chaincode would validate every hash submission and check it against the previous submission. If a “hash mismatch” occurred, Hyperledger would issue an “event” that the target agent would detect, and the agent would notify a team member by Short Messaging Service (SMS). An example of the initial Hyperledger test assets is illustrated below in Figure 4, which is a screen capture of the Hyperledger assets. Note - original asset names as well as organizations have been redacted and replaced with generic names of “TEST-ORG#” and “TEST-ASSET” or “ASSET-#” in further screen captures.

For this use case, the Hyperledger was populated with three test assets with inaccurate “hash” values: “TEST-ASSET1,” “TEST-ASSET-2,” and “TEST-ASSET-3”.

Figure 4. Initial Hyperledger test assets.

The installed agent would track the "TEST-ASSET-1" asset. The HELICS agent was then started on the target asset "TEST-ASSET-1" and the process began.

2.2.5.1 Process

The process consisted of executing the HELICS agent, which would run an appropriate STIG Analysis tool based on the host's operating system (CyberKnight, or SCC). An example HELICS agent starting is shown in Figure 5.

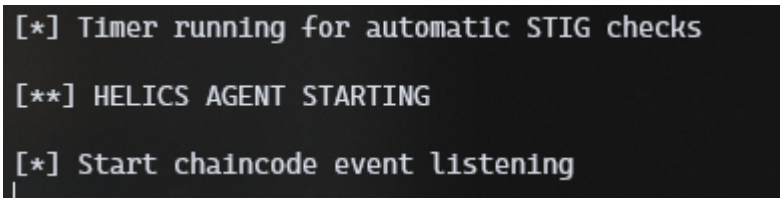


Figure 5. Example HELICS agent starting.

The agent then analyzed the target asset and created an appropriate DISA STIG checklist for any STIGs which were applicable to the test system. The STIG checklist was parsed, and a list of hashes were created for each unique STIG check. Those individual hashes were then compiled into a single

baseline hash, and then submitted to the Hyperledger. An example of the HELICS agent completing STIG analysis and updating the Hyperledger is shown in Figure 6.

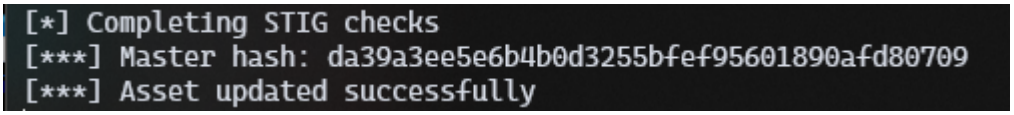


Figure 6. HELICS agent completing STIG analysis and updating the Hyperledger.

The Hyperledger then revalidated the baseline hash value every five minutes. No action was taken if the hash value was a match, and the hash value was written to the Hyperledger. If the submitted hash value had changed from the previous hash, an event was triggered (hash mismatch) and the agent would respond by initiating a smart contract. The "bad" hash was written to the ledger to maintain an immutable record of events (useful for auditing, forensics, etc.)

In order to simulate a "bad" hash, a custom bash script was used to manually edit the hash value prior to submission to the Hyperledger. Manually changing the hash in the Hyperledger would issue a hash-mismatch event and trigger the smart contract.

2.2.5.1.1 Measurement Method

Measurements were captured by:

- Command Line Interface (CLI) with the target asset to capture results of the STIG analysis and errors when interacting with the Hyperledger via screenshots.
- Hyperledger Explorer project (GitHub - hyperledger-labs/blockchain-explorer) which allowed for inspection of each block and the data contained within it, in order to ensure target test data was accurately being captured and written to the Hyperledger.

2.3 MATERIALS

The test was installed into a 64-bit Ubuntu 20.04 Linux Server with Linux kernel version 5.4.0-153 inside a VMWare Workstation 16 Pro Virtual Machine consisting of 1 virtual CPU and 8 virtual cores, 16 GB of RAM and 250GB of disk space.

The Hyperledger Fabric test-network was implemented using Docker version 24.0.0 and Docker Compose version 1.29.2. The installed agent was programmed and compiled in Go version 1.20.2. CyberKnight was also compiled using Go version 1.20.2. SCC was provided as binary. Bash script tools were written using GNU Bash version 5.0.17

The following software tools and/or repositories were involved in this use case:

(Appendix A provides more detailed information)

- VMWare Workstation 16 Professional®
- Ubuntu 20.04 x64 Server®
- Hyperledger Fabric test-network®
- The programming language Go
- Docker and Docker Compose®

- Hyperledger Fabric Explorer®
- Grafana & CAdvisor®
- Custom agent and bash scripts written by the team

This page is intentionally blank.

3. RESULTS

3.1 RESULTS

The use case produced a successful test of the HELICS agent. The tool provided successful detection and validation of the test system's secure baseline via continuous monitoring of its hash value using Hyperledger Fabric. The implementation of smart contracts was successful for providing real-time notification for changes to the system's secure baseline. When the target system or the Hyperledger was manually manipulated, the agent and "chaincode" detected and responded automatically to the changes.

The installed agent ran STIG automation tools every five minutes, analyzing the target system for secure configuration changes, and when changes were detected, a real time notification was successfully sent.

- CyberKnight® is the STIG automation tool used to perform STIG analysis during the test.
- Notifications were sent out via SMS using a trial account of the 'Twilio®' service.

3.1.1 Key Findings

- The agent ran in the background and was not resource intensive. Figure 7 shows Hyperledger & HELICS agent resource usage. Even while performing STIG audits, resource usage was minimal:

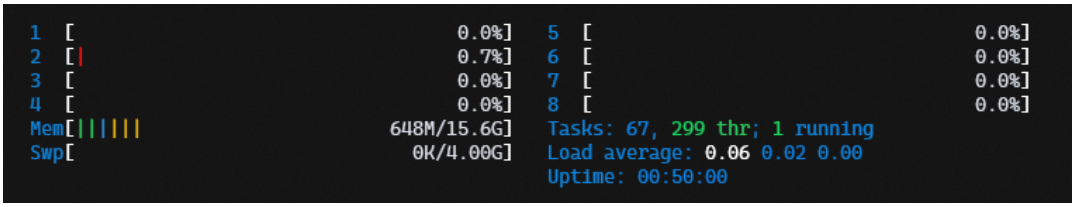


Figure 7. Hyperledger & HELICS agent resource usage.

- The agent ran in the background (Figure 8 shows a screen capture showing HELICS agent running) and continuously updated the Hyperledger:

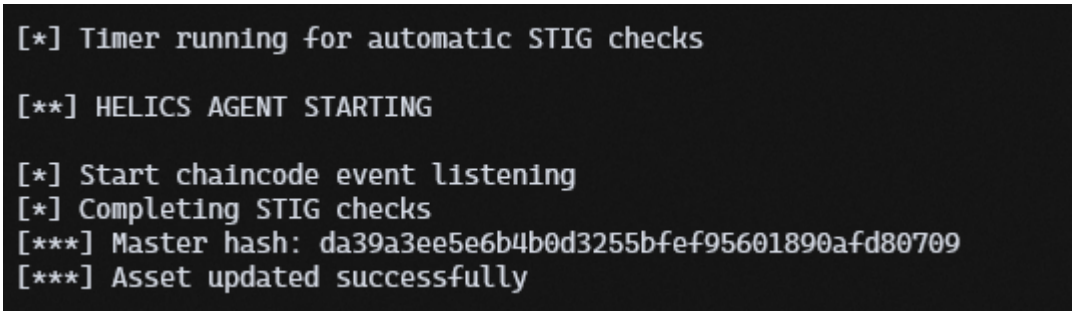


Figure 8. HELICS agent running.

Figure 9 shows the HELICS agent producing steady-state hashes.

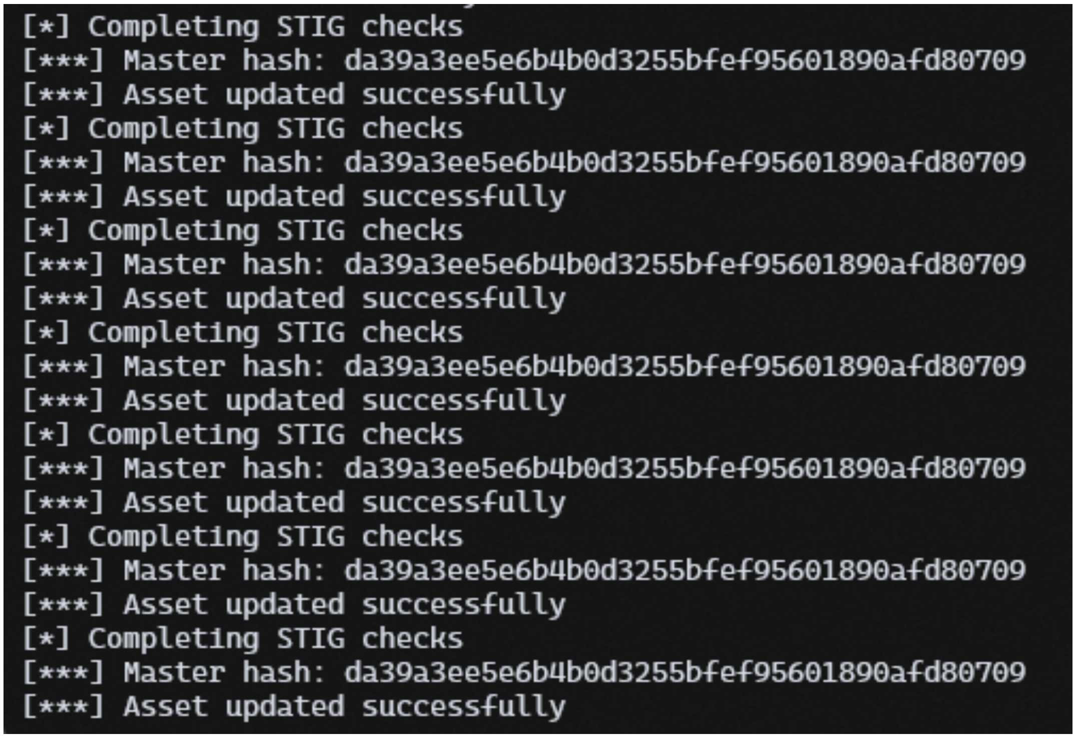
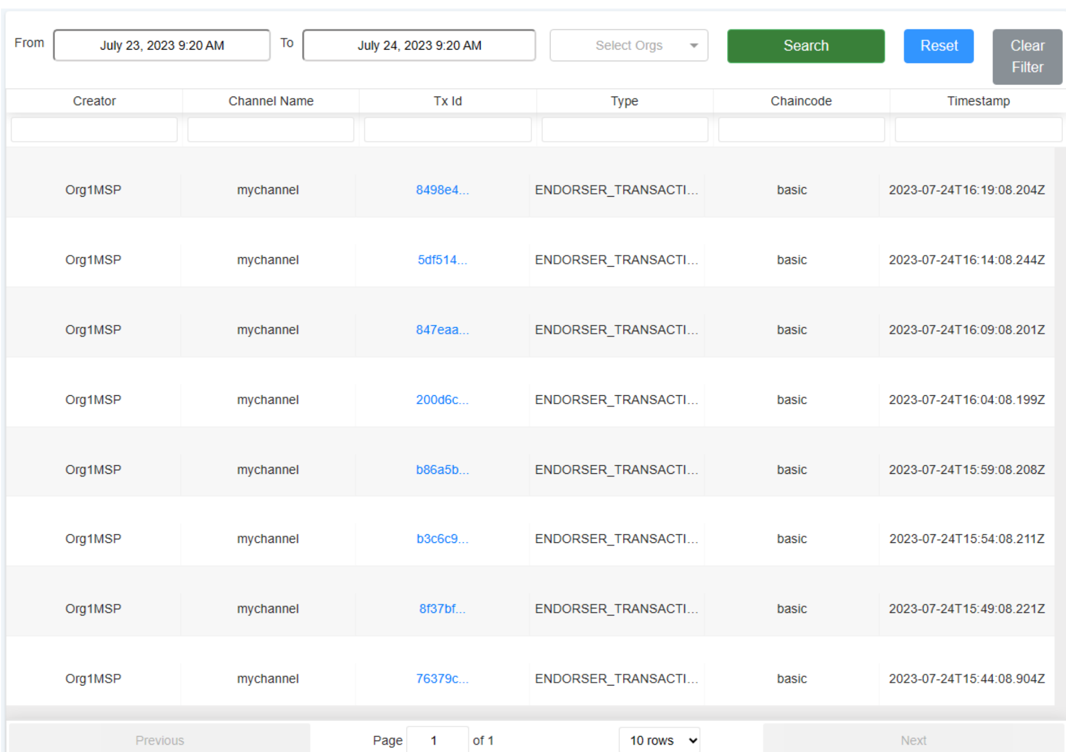


Figure 9. HELICS agent producing steady-state hashes.

The Hyperledger functioned as anticipated and tracked asset data properly. Using Hyperledger Explorer each transaction and block could be inspected. Figure 10 shows a Hyperledger Blocks list.



From July 23, 2023 9:20 AM To July 24, 2023 9:20 AM Select Orgs Search Reset Clear Filter

Creator	Channel Name	Tx Id	Type	Chaincode	Timestamp
Org1MSP	mychannel	8498e4...	ENDORSER_TRANSACTI...	basic	2023-07-24T16:19:08.204Z
Org1MSP	mychannel	5df514...	ENDORSER_TRANSACTI...	basic	2023-07-24T16:14:08.244Z
Org1MSP	mychannel	847eaa...	ENDORSER_TRANSACTI...	basic	2023-07-24T16:09:08.201Z
Org1MSP	mychannel	200d6c...	ENDORSER_TRANSACTI...	basic	2023-07-24T16:04:08.199Z
Org1MSP	mychannel	b86a5b...	ENDORSER_TRANSACTI...	basic	2023-07-24T15:59:08.208Z
Org1MSP	mychannel	b3c6c9...	ENDORSER_TRANSACTI...	basic	2023-07-24T15:54:08.211Z
Org1MSP	mychannel	8f37bf...	ENDORSER_TRANSACTI...	basic	2023-07-24T15:49:08.221Z
Org1MSP	mychannel	76379c...	ENDORSER_TRANSACTI...	basic	2023-07-24T15:44:08.904Z

Previous Page 1 of 1 10 rows Next

Figure 10. List of Hyperledger Blocks.

Figure 11 shows an example of a specific Hyperledger block transaction.

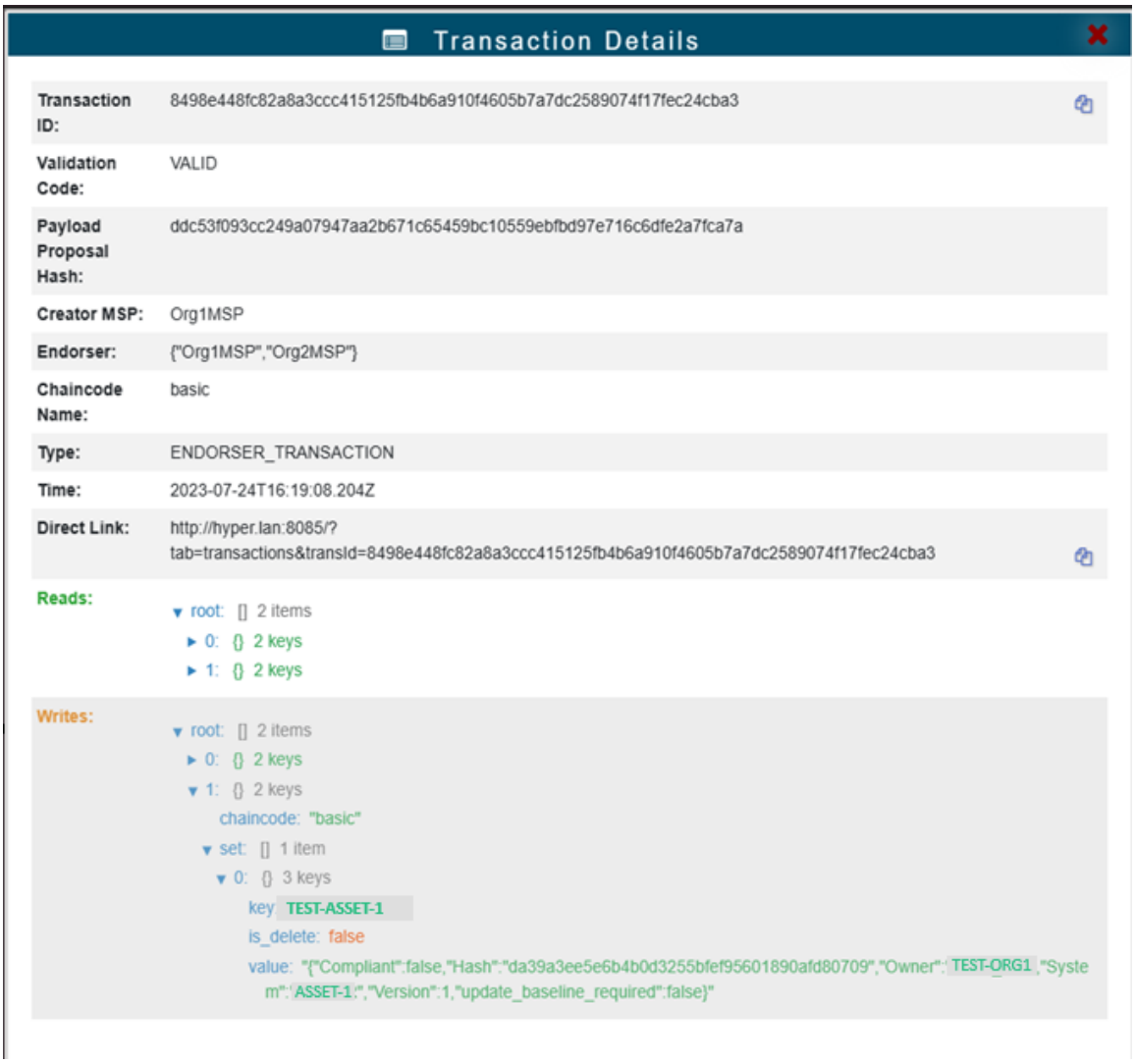


Figure 11. Specific Hyperledger block transaction.

- The Hyperledger also validated and issued "event" data when stimulated properly to the deployed agents. Figure 12 shows an example of manually manipulating the Hyperledger.

```
jallphin@hyper:/opt/hyperledger/helics/toolage$ ./update-asset.sh ASSET-1 TEST-HASH-INVALID false TEST-ORG1 false
2023-07-24 16:24:13.907 UTC 0001 INFO [chaincodeCmd] chaincodeInvokeOrQuery -> Chaincode invoke successful. result:
 status:200
```

Figure 12. Manually manipulating the Hyperledger.

- When a "hash-mismatch" event did occur, the HELICS agent successfully took immediate, programmable action. Figure 13 shows an example hash-mismatch, triggering an SMS alert.

```
[!!!] Hash Mismatch event received: {"Compliant":false,"Hash":"da39a3ee5e6b4b0d3255bfef95601890afd80709","Owner":
"TEST-ORG1","System":"ASSET-1","Version":1,"update_baseline_required":false}
[***] SMS sent successfully
```

Figure 13. HELICS agent notices hash-mismatch and takes action.

- A SMS screenshot of HELICS notification is shown in Figure 14.

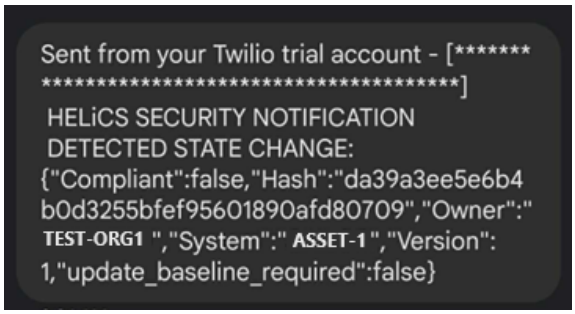


Figure 14. SMS screenshot of HELICS notification.

- Historical results were available which provide traceability and a complete record of secure configuration compliance when utilized. Figure 15 shows has the example "TEST-HASH-INVALID" was manually invoked onto the Hyperledger. Five minutes later, the agent re-scanned the asset and wrote the correct hash to the Hyperledger:


```
[
  {
    "txId": "e6e7c211f180d9b030448bb35165e42b974591f63466b3abf39d0689c7117a28",
    "value": {
      "Compliant": false,
      "Hash": "da39a3ee5e6b4b0d3255bfef95601890afd80709",
      "Owner": TEST-ORG1 ,
      "System": ASSET-1 ,
      "Version": 1,
      "update_baseline_required": false
    },
    "timestamp": "2023-07-24 16:29:08"
  },
  {
    "txId": "6161e65eb37c83bd448ebec1000e569e3497ac0b5946100aba6bf05e3bfe7a0d",
    "value": {
      "Compliant": false,
      "Hash": "TEST-HASH-INVALID",
      "Owner": TEST-ORG1 ,
      "System": TEST-ASSET-2 ,
      "Version": 1,
      "update_baseline_required": false
    },
    "timestamp": "2023-07-24 16:24:13"
  },
```


Figure 15. Historical record of asset on Hyperledger.

4. DISCUSSION

4.1 MAJOR FINDINGS

Overall, this use case proved that greater efficiency and efficacy can be achieved for Continuous Monitoring and Vulnerability Management by implementing automated STIG analysis combined with a properly configured Hyperledger to actively monitor a system's secure baseline. Adding smart contracts provides additional capability for automating incident response activities. However, there are technical challenges which need to be overcome to fully capitalize on the robust potential benefits of HELICS if it were successfully implemented in a DoD enterprise environment.

4.1.1 Finding Details

During research it was discovered that the "Finding Details" section of a completed STIG Benchmark oftentimes does not retain the same hash value due to issues related to the file system and permissions. In a running information system, which is actively ingesting and processing data, there are dynamic file system modifications.

The notable exception to this issue is operating systems which are configured to be "immutable." An "immutable OS (also known as Immutable Infrastructure or Immutable Deployment) is an operating system designed to be unchangeable and read-only. This means that once the operating system has been installed, the system files and directories cannot be modified. Any changes made to the system are temporary and lost when the system is rebooted" (Giacinto, 2023). DoD does not actively deploy these Operating Systems, so they were not included in the research.

In order to overcome the challenge of achieving a steady state secure configuration for this particular use case, the problematic STIG checks were removed from the hashing algorithm. For the largest STIGs used, this amounted to approximately 10% of the total STIG checks. For any future implementation of HELICS, each STIG which checks for file system modifications will need to be tuned so a "steady-state" configuration baseline can be achieved.

4.1.2 Implications

There are numerous implications for the HELICS agent:

- Improved Continuous Monitoring effectiveness and efficiency – This use case proves that HELICS, when properly implemented, can meet the challenge of the NIST SP800-137, "When possible, organizations look for automated solutions to lower costs, enhance efficiency, and improve the reliability of monitoring security-related information. Security is implemented through a combination of people, processes, and technology. The automation of information security deals primarily with automating aspects of security that require little human interaction. Automated tools are often able to recognize patterns and relationships that may escape the notice of human analysts, especially when the analysis is performed on large volumes of data" (Dempsey, et al., 2011). The DoD's current Continuous Monitoring implementation requires countless personnel hours for actively and manually monitoring & reviewing systems for vulnerabilities. HELICS can significantly reduce costs for programs to properly monitor systems. Additionally, automation using HELICS has the potential to reduce costs associated with Risk Management Framework re-evaluation for systems that do not require fundamental changes to its architecture, streamlining re-authorizations.
- Versatile functionality via a deployable and distributed set up – Although this use case was only used to track the security configuration of an asset, Hyperledger is capable and effective

for tracking any data. This provides the ability for system owners to track and query information in a global environment in real time, and it supports monitoring for data in a Disrupted, Intermittent, and Limited (DIL) environment.

- Open Source and low cost – HELICS's specific implementation utilizing Hyperledger Fabric is open sourced under the Apache 2.0 License. This provides the DoD the freedom to patent any technology developed using Hyperledger Fabric and use it for free.

4.1.2.1.1 Future Research

More research is needed to ascertain how effectively Hyperledger and STIG Analysis can run in an enterprise environment. Specifically:

- Scalability in an Enterprise environment.
- Keeping the installed agent's STIGs updated so it is scanning with the most up-to-date STIG Benchmarks.
- Proper real-time notification and/or action taken sequencing.
- Configuration into complex deployable and battlefield scenarios.

4.2 UNANSWERED QUESTIONS

Some of the technical challenges presented during this use case require permanent solutions to be implemented in order for HELICS to serve as a legitimate capability in an enterprise environment. The most significant example is the challenge of achieving a "steady state" secure configuration when it comes to STIG Analysis. Details pertaining to this finding are provided in Section 4.1.1. More research is needed to identify the most effective solution for achieving a steady state for a system.

Additionally, there were fiscal and environmental constraints to testing HELICS at a large scale for this use case. Due to the limited scope and funding for this use case, testing was restricted to a small virtual environment using four Docker containers. Based on previous results from implementations in private industry, it is believed it will function at significantly larger scales. For this use case however, HELICS remains untested for that level of scalability.

5. CONCLUSIONS

The potential benefits for HELICS are numerous and significant. Combining automated STIG analysis with Hyperledger Fabric can provide a highly functional and efficient tool for Continuous Monitoring and Vulnerability Management. The tool has a lower cost than existing patch management solutions used by the DoD as it leverages existing open-source software. With the implementation of Smart Contracts, this tool provides a means for automated response which exceeds the DoD's currently deployed capabilities, and there are several additional potential benefits worth exploring further. HELICS has potential to address shortfalls in supply-chain oversight, zero-day attacks, and incident response – just to name a few. With all this potential, HELICS demands further research into this burgeoning field of technology.

5.1 RECOMMENDATIONS

The success of this use case test suggest that it is worth pursuing funding for additional research and testing in a lab environment in order to investigate enterprise scalability. This would offer the opportunity to pursue new and permanent solutions to the "steady-state" problem, and to explore the numerous possibilities regarding the automated incident response capabilities of Smart Contracts in situations involving legitimate secure configuration changes – whether malicious or not.

This page is intentionally blank.

REFERENCES

Centieiro, H. (2021, April 23). *Public Blockchains vs Private Blockchains*. Retrieved from Medium.com: https://levelup.gitconnected.com/public-blockchains-vs-private-blockchains-96514dfae3a7

Dempsey, K., Chawla, N. S., Johnson, A., Johnston, R., Jones, A. C., Orebaugh, A., . . . Stine, K. (2011, September). Information Security Continuous Monitoring (ISCM) for Federal Information Systems and Organizations. *NIST Special Publication 800–137*. Gaithersburg, MD, USA: National Institute of Standards and Technology.

FireEye Mandiant. (2021). *FireEye Mandiant Services Special Report, M-Trends 2021.* Milpitas, CA: FireEye, Inc.

Giacinto, E. D. (2023, March 22). *Understanding Immutable Linux OS: Benefits, Architecture, and Challenges*. Retrieved from Kairos.io: https://kairos.io/blog/2023/03/22/understanding-immutable-linux-os-benefits-architecture-and-challenges/

Hyperledger Fabric. (2023, July 27). *The Ordering Service*. Retrieved from hyperledger-fabric.readthedocs.io: https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html#raft

Hyperledger Foundation. (2023, July 27). *About Hyperledger Foundation*. Retrieved from Hyperledger.org: https://www.hyperledger.org/about

IBM. (2023, July 27). *What are smart contracts on blockchain?* Retrieved from IBM.com: https://www.ibm.com/topics/smart-contracts

U.S. Government Accountability Office (GAO). (2021, April 22). *SolarWinds Cyberattack Demands Significant Federal and Private-Sector Response (infographic)*. Retrieved from GAO.gov: https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic

.

This page is intentionally blank.

APPENDIX A
HELICS REFERENCE CODE AS IMPLEMENTED LISTING

A.1 OVERVIEW

Below is a listing of HELICS code screen captures examples included in this appendix:

HELICS Agent

The following pages A-2 through A-9 are the code for the HELICS Agent.

```
package main

import (
    "context"
    "crypto/x509"
    "fmt"
    "os"
    "path"
    "time"
    "os/exec"
    "strings"
    "io/ioutil"
    "strconv"

    "github.com/hyperledger/fabric-gateway/pkg/client"
    "github.com/hyperledger/fabric-gateway/pkg/identity"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "github.com/twilio/twilio-go"
    twilioApi "github.com/twilio/twilio-go/rest/api/v2010"

)

const (
    channelName   = "mychannel"
    chaincodeName = "basic"
    mspID         = "Org1MSP"
    cryptoPath    = "../../test-
network/organizations/peerOrganizations/org1.example.com"
    certPath      = cryptoPath +
"/users/User1@org1.example.com/msp/signcerts/User1@org1.example.com-cert.pem"
    keyPath       = cryptoPath + "/users/User1@org1.example.com/msp/keystore/"
    tlsCertPath   = cryptoPath + "/peers/peer0.org1.example.com/tls/ca.crt"
    peerEndpoint  = "localhost:7051"
    gatewayPeer   = "peer0.org1.example.com"
)

func main() {
    clientConnection := newGrpcConnection()
    defer clientConnection.Close()
```

```
    id := newIdentity()
    sign := newSign()

    gateway, err := client.Connect(
        id,
        client.WithSign(sign),
        client.WithClientConnection(clientConnection),
        client.WithEvaluateTimeout(5*time.Second),
        client.WithEndorseTimeout(15*time.Second),
        client.WithSubmitTimeout(5*time.Second),
        client.WithCommitStatusTimeout(1*time.Minute),
    )
    if err != nil {
        panic(err)
    }
    defer gateway.Close()

    network := gateway.GetNetwork(channelName)
    contract := network.GetContract(chaincodeName)
    system := "TEST-ASSET-1"

    // Context used for event listening
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    // Define the interval for STIG checks and ledger updates
    checkInterval := 5 * time.Minute

    // Create a ticker to trigger STIG checks and ledger updates periodically
    ticker := time.NewTicker(checkInterval)
    defer ticker.Stop()

    go func() {
        for range ticker.C {
            performSTIGChecksAndUpdateLedger(contract, system)
            if err != nil {
                fmt.Printf("[**] Error performing STIG checks and updating
ledger: %s\n", err)
            }
        }
    }()

    fmt.Println("\n[*] Timer running for automatic STIG checks")
```

```
    // Listen for events emitted by the basic-asset-transfer chaincode
    startChaincodeEventListening(ctx, network, contract, system)
    select {} // Prevent the main function from exiting
}

// BETA LISTENER CODE
func startChaincodeEventListening(ctx context.Context, network *client.Network,
contract *client.Contract, system string) {
    fmt.Println("\n[*] Start chaincode event listening")

    events, err := network.ChaincodeEvents(ctx, chaincodeName)
    if err != nil {
        panic(fmt.Errorf("[**] failed to start chaincode event listening: %w",
err))
    }

    go func() {
        for event := range events {
            if event.EventName == "HashMismatch" {
                fmt.Printf("\n[!!!] Hash Mismatch event received: %s\n",
string(event.Payload))

                // Send SMS when a hash mismatch occurs
                //smsBody := fmt.Sprintf("Hash Mismatch detected: %s",
string(event.Payload))
                smsBody :=
fmt.Sprintf("[*****************************************]\n HELiCS SECURITY
NOTIFICATION\n DETECTED STATE CHANGE: %s\n\n", string(event.Payload))
                phoneNumber := "REPLACE_WITH_VALID_CELL_NUMBER"

                err := sendSMS(phoneNumber, smsBody)
                if err != nil {
                    fmt.Printf("[**] Error sending SMS: %s\n", err)
                } else {
                    fmt.Println("[***] SMS sent successfully")
                }
            } else if event.EventName == "UpdateBaselineRequiredSet" {
                fmt.Printf("\n[!!!] UpdateBaselineRequiredSet event received:
%s\n", string(event.Payload))
                updateBaselineHash(contract, system)

            }
        }
    }()
```

```
}
func newGrpcConnection() *grpc.ClientConn {
    certificate, err := loadCertificate(tlsCertPath)
    if err != nil {
        panic(err)
    }

    certPool := x509.NewCertPool()
    certPool.AddCert(certificate)
    transportCredentials := credentials.NewClientTLSFromCert(certPool,
gatewayPeer)

    connection, err := grpc.Dial(peerEndpoint,
grpc.WithTransportCredentials(transportCredentials))
    if err != nil {
        panic(fmt.Errorf("[**] failed to create gRPC connection: %w", err))
    }

    return connection
}

func newIdentity() *identity.X509Identity {
    certificate, err := loadCertificate(certPath)
    if err != nil {
        panic(err)
    }

    id, err := identity.NewX509Identity(mspID, certificate)
    if err != nil {
        panic(err)
    }

    return id
}

func loadCertificate(filename string) (*x509.Certificate, error) {
    certificatePEM, err := os.ReadFile(filename)
    if err != nil {
        return nil, fmt.Errorf("[**] failed to read certificate file: %w", err)
    }
    return identity.CertificateFromPEM(certificatePEM)
}

func newSign() identity.Sign {
```

```
    files, err := os.ReadDir(keyPath)
    if err != nil {
        panic(fmt.Errorf("[**] failed to read private key directory: %w", err))
    }
    privateKeyPEM, err := os.ReadFile(path.Join(keyPath, files[0].Name()))

    if err != nil {
        panic(fmt.Errorf("[**] failed to read private key file: %w", err))
    }

    privateKey, err := identity.PrivateKeyFromPEM(privateKeyPEM)
    if err != nil {
        panic(err)
    }

    sign, err := identity.NewPrivateKeySign(privateKey)
    if err != nil {
        panic(err)
    }

    return sign
}

//func performSTIGChecksAndUpdateLedger(contract *client.Contract) error {
func performSTIGChecksAndUpdateLedger(contract *client.Contract, system string) {
    // Perform STIG checks
        fmt.Println("[*] Completing STIG checks")
    //scriptPath := "/opt/hyperledger/CyberKnight/CyberKnight.sh"
    //cmd := exec.Command("bash", scriptPath)
        cmd := exec.Command("/bin/bash",
"/opt/hyperledger/CyberKnight/CyberKnight.sh")
    // Discard the output and error streams
        cmd.Stdout = ioutil.Discard
        cmd.Stderr = ioutil.Discard

        err := cmd.Run()
        if err != nil {
           fmt.Printf("[**] failed to execute STIG checks script: %w", err)
       return
        }

    // Update the ledger with the new hash
    // Read the contents of the master_hash.txt file
```

```
        content, err :=
os.ReadFile("/opt/hyperledger/CyberKnight/current_master_hash.txt")
           if err != nil {
               fmt.Printf("[**] Error reading master_hash.txt: %s\n", err)
               return
        }

    // Convert the content to a string and split it into parts using the ": "
separator
        parts := strings.Split(string(content), ": ")

    // Extract the master hash (first part) and timestamp (second part)
        masterHash := parts[0]
        // timestamp := parts[1]
    // Define the other parameters for the updateAsset function
        //system := "SAILOR"     // Replace with the appropriate value
        compliant := false           // Replace with the appropriate value
        owner := "TEST_ORG1"       // Replace with the appropriate value
        version := 1                // Replace with the appropriate value
    UpdateBaselineRequired := false // Not updating the baseline hash right now
        fmt.Printf("[***] Master hash: %s\n", masterHash)

    // Call the updateAsset function in the chaincode
    _, err = contract.SubmitTransaction("UpdateAsset", system, masterHash,
strconv.FormatBool(compliant), owner, strconv.Itoa(version),
strconv.FormatBool(UpdateBaselineRequired))
    if err != nil {
        fmt.Printf("[**] Error updating asset: %s\n", err)
        return
    }
    fmt.Println("[***] Asset updated successfully")
    return
}

func sendSMS(to, body string) error {
    accountSid := "REPLACE_WITH_ACCOUNT_SID"
    authToken := "REPLACE_WITH_AUTH_TOKEN"
    from := "REPLACE_WITH_TWILIO_NUMBER"

    client := twilio.NewRestClientWithParams(twilio.ClientParams{
        Username: accountSid,
        Password: authToken,
    })
```

```
    params := &twilioApi.CreateMessageParams{}
    params.SetTo(to)
    params.SetFrom(from)
    params.SetBody(body)

    _, err := client.Api.CreateMessage(params)

    if err != nil {
        return fmt.Errorf("failed to send SMS: %v", err)
    }

    return nil
}


func updateBaselineHash(contract *client.Contract, system string) {
    fmt.Println("[*] Updating baseline hash")

    // Perform STIG checks
    cmd := exec.Command("/bin/bash",
"/opt/hyperledger/CyberKnight/CyberKnight.sh")
    cmd.Stdout = ioutil.Discard
    cmd.Stderr = ioutil.Discard

    err := cmd.Run()
    if err != nil {
        fmt.Printf("[**] Failed to execute STIG checks script: %s\n", err)
        return
    }

    // Read the contents of the master_hash.txt file
    content, err :=
os.ReadFile("/opt/hyperledger/CyberKnight/current_master_hash.txt")
    if err != nil {
        fmt.Printf("[**] Error reading master_hash.txt: %s\n", err)
        return
    }

    // Extract the new master hash
    parts := strings.Split(string(content), ": ")
    newMasterHash := parts[0]

    // Update the asset on the ledger
```

```
    //_, err = contract.SubmitTransaction("UpdateHash", system, newMasterHash,
"false", "AssetOwner", "1")
    _, err = contract.SubmitTransaction("UpdateHash", system, newMasterHash)
    if err != nil {
        fmt.Printf("[**] Error updating asset: %s\n", err)
        return
    }

    fmt.Printf("[***] Asset updated successfully with new baseline hash: %s\n",
newMasterHash)

    // Reset the UpdateBaselineRequired flag
    _, err = contract.SubmitTransaction("SetUpdateBaselineRequired", system,
"false")
    if err != nil {
        fmt.Printf("[**] Error resetting UpdateBaselineRequired flag: %s\n", err)
        return
    }

    fmt.Println("[***] UpdateBaselineRequired flag reset")
}
```

HELICS Initialize Chaincode Script

The following pages A-10 through A-13 describe the HELICS Initialization Chaincode script.

```
#!/bin/bash
# Change to working directory
cd /opt/hyperledger/fabric-samples/test-network

# Set version
VERSION=1.0

# Check for optional command arguments to manually increment label + sequence for
chaincode updates, if not present default:
# 1 should be label name, 2 should be sequence number.
if [ -z "$1" ] && [ -z "$2" ]
then
        set -- "ckcc_1" "1"
fi

#Export path so peer commands work in the cli
export PATH=/opt/hyperledger/fabric-samples/bin:$PATH
export FABRIC_CFG_PATH=/opt/hyperledger/fabric-samples/test-network/../config/

## Check to see if the channel is existant and available
CHANNEL_EXISTS=$(peer channel list | grep mychannel)
if [ "$CHANNEL_EXISTS" != "mychannel" ]
  then
    echo "[*] Channel does not exist, creating."
    echo "[*] Installing the channel 'mychannel' with network.sh"
    ./network.sh createChannel -c mychannel
  else
    echo "[*] Channel exists, skipping."
fi
sleep 1.5

echo "[*] Checking peer version"
peer version
sleep 3

echo "[*] Packaging up chaincode $1"
peer lifecycle chaincode package ckcc.tar.gz --path
/opt/hyperledger/helics/chaincode --lang golang --label $1
sleep 3
```

```
#First Org
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/hyperledger/fabric-samples/test-
network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example
.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabric-samples/test-
network/organizations/peerOrganizations/org1.example.com/users/Admin@org1.example
.com/msp
export CORE_PEER_ADDRESS=localhost:7051

#Install chaincode package
echo "[*] Installing chaincode package in first org"
peer lifecycle chaincode install ckcc.tar.gz
sleep 3
#Second Org
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_LOCALMSPID="Org2MSP"
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/hyperledger/fabric-samples/test-
network/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example
.com/tls/ca.crt
export CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabric-samples/test-
network/organizations/peerOrganizations/org2.example.com/users/Admin@org2.example
.com/msp
export CORE_PEER_ADDRESS=localhost:9051
echo "[*] Installing chaincode package in second org"
peer lifecycle chaincode install ckcc.tar.gz
sleep 3

#Approve chaincode definition
echo "[*] Approving chaincode definition and extracting package ID"
peer lifecycle chaincode queryinstalled &> ./package_id.txt

#Saving package ID as environment variable
#Will need to figure out how to automate this part

export CC_PACKAGE_ID=$(grep $1 package_id.txt | cut -d " " -f3 | rev | cut -c2- |
rev)

peer lifecycle chaincode approveformyorg -o localhost:7050 --
ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic
--version 1.0 --package-id $CC_PACKAGE_ID --sequence $2 --tls --cafile
"${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.c
om/msp/tlscacerts/tlsca.example.com-cert.pem"
```

```
sleep 3

echo "[*] Committing changes to Org1"
#commit with org1
export CORE_PEER_LOCALMSPID="Org1MSP"
export CORE_PEER_MSPCONFIGPATH=/opt/hyperledger/fabric-samples/test-
network/organizations/peerOrganizations/org1.example.com/users/Admin@org1.example
.com/msp
export CORE_PEER_TLS_ROOTCERT_FILE=/opt/hyperledger/fabric-samples/test-
network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example
.com/tls/ca.crt
export CORE_PEER_ADDRESS=localhost:7051
peer lifecycle chaincode approveformyorg -o localhost:7050 --
ordererTLSHostnameOverride orderer.example.com --channelID mychannel --name basic
--version 1.0 --package-id $CC_PACKAGE_ID --sequence $2 --tls --cafile
"${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.c
om/msp/tlscacerts/tlsca.example.com-cert.pem"
sleep 3

#Commiting chaincode definition to the channel
echo "[*] Committing chaincode definitions to the channel"
peer lifecycle chaincode checkcommitreadiness --channelID mychannel --name basic
--version 1.0 --sequence $2 --tls --cafile
"${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.c
om/msp/tlscacerts/tlsca.example.com-cert.pem" --output json

#Commit chaincode to the channel
peer lifecycle chaincode commit -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --channelID mychannel --name basic --version 1.0 --sequence
$2 --tls --cafile
"${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.c
om/msp/tlscacerts/tlsca.example.com-cert.pem" --peerAddresses localhost:7051 --
tlsRootCertFiles
"${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example
.com/tls/ca.crt" --peerAddresses localhost:9051 --tlsRootCertFiles
"${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example
.com/tls/ca.crt"
sleep 3
#Verify chaincode has been committed
echo "[*] Verifying chaincode has been comitted"
peer lifecycle chaincode querycommitted --channelID mychannel --name basic

#At this point the chaincode should be installed and can be invoked
```

```
echo "[*] Invoking the chaincode so we can run commands."
peer chaincode invoke -o localhost:7050 --ordererTLSHostnameOverride
orderer.example.com --tls --cafile
"${PWD}/organizations/ordererOrganizations/example.com/orderers/orderer.example.c
om/msp/tlscacerts/tlsca.example.com-cert.pem" -C mychannel -n basic --
peerAddresses localhost:7051 --tlsRootCertFiles
"${PWD}/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example
.com/tls/ca.crt" --peerAddresses localhost:9051 --tlsRootCertFiles
"${PWD}/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example
.com/tls/ca.crt" -c '{"function":"InitLedger","Args":[]}'

echo "[*] If you don't see any serious errors, it appears we are good to go."
```

HELICS Chaincode

The following pages A-14 through A-22 describe the HELICS Chaincode.

```go
package main

import (
    "encoding/json"
    "fmt"
    "log"

    "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

// SmartContract provides functions for managing an Asset
type SmartContract struct {
    contractapi.Contract
}

// Asset describes basic details of what makes up a simple asset
type Asset struct {
    // Must be in alphabetic order to achieve determinism accross languages
    Compliant bool   `json:"Compliant"`
    Hash      string `json:"Hash"`
    Owner     string `json:"Owner"`
    System    string `json:"System"`
    Version   int    `json:"Version"`
    UpdateBaselineRequired bool `json:"update_baseline_required"`
}

// AssetHistory struct allows us to iterate over the history of an asset
type AssetHistory struct {
    TxId      string `json:"txId"`
    Value     Asset  `json:"value"`
    Timestamp string `json:"timestamp"`
}

// InitLedger adds a base set of assets to the ledger
// This works as intended
func (s *SmartContract) InitLedger(ctx contractapi.TransactionContextInterface)
error {
    assets := []Asset{
```

```
        {System: "TEST-ASSET-1", Hash:
"39cf4b5cc96e4747e87faa4059b7c68d90a82cfc", Compliant: true, Owner: "TEST-ORG1",
Version: 1, UpdateBaselineRequired: false},
        {System: "TEST-ASSET-2", Hash:
"e2289b0d196d46fd1a6cee86dcde81adbd9919da", Compliant: false, Owner: "TEST-ORG2",
Version: 1, UpdateBaselineRequired: false},
        {System: "TEST-ASSET-3", Hash:
"1c02be421de8d371213494f56ac6ec3698791349", Compliant: true, Owner: "TEST-ORG3",
Version: 1, UpdateBaselineRequired: false},
    }

    for _, asset := range assets {
        assetJSON, err := json.Marshal(asset)
        if err != nil {
            return err
        }

        err = ctx.GetStub().PutState(asset.System, assetJSON)
        if err != nil {
            return fmt.Errorf("failed to put to world state. %v", err)
        }
    }

    return nil
}

// GetAssetHistory iterates over the ledger history of asset allowing us to see
changes to the asset.
func (s *SmartContract) GetAssetHistory(ctx
contractapi.TransactionContextInterface, system string) ([]AssetHistory, error) {
    // Obtain the transaction history iterator for the given asset key
    resultsIterator, err := ctx.GetStub().GetHistoryForKey(system)
    if err != nil {
        return nil, fmt.Errorf("failed to get asset history: %v", err)
    }
    defer resultsIterator.Close()

    var history []AssetHistory

    // Iterate through the transaction history and construct the AssetHistory
objects
    for resultsIterator.HasNext() {
        response, err := resultsIterator.Next()
        if err != nil {
```

```
            return nil, fmt.Errorf("failed to read from asset history iterator:
%v", err)
        }

        // Check if the value is empty and skip the iteration
        if len(response.Value) == 0 {
            continue
        }

        var asset Asset
        err = json.Unmarshal(response.Value, &asset)
        if err != nil {
            return nil, fmt.Errorf("failed to unmarshal asset JSON: %v", err)
        }

        // Convert the timestamppb.Timestamp to time.Time and format it as a
human-readable string (e.g., "2006-01-02 15:04:05")
        timestamp := response.Timestamp.AsTime().Format("2006-01-02 15:04:05")

        assetHistory := AssetHistory{
            TxId:      response.TxId,
            Value:     asset,
            Timestamp: timestamp,
        }

        history = append(history, assetHistory)
    }

    return history, nil
}


// CreateAsset issues a new asset to the world state with given details.
// Works as intended
func (s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface,
system string, hash string, compliant bool, owner string, version int,
updatebaselinerequired bool) error {
    exists, err := s.AssetExists(ctx, system)
    if err != nil {
        return err
    }
    if exists {
        return fmt.Errorf("the asset %s already exists", system)
    }
```

```
    asset := Asset{
        System:    system,
        Hash:      hash,
        Compliant: compliant,
        Owner:     owner,
        Version:   version,
        UpdateBaselineRequired: updatebaselinerequired,
    }
    assetJSON, err := json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(system, assetJSON)
}

// ReadAsset returns the asset stored in the world state with given system.
// Works as intended
func (s *SmartContract) ReadAsset(ctx contractapi.TransactionContextInterface,
system string) (*Asset, error) {
    assetJSON, err := ctx.GetStub().GetState(system)
    if err != nil {
        return nil, fmt.Errorf("failed to read from world state: %v", err)
    }
    if assetJSON == nil {
        return nil, fmt.Errorf("the asset %s does not exist", system)
    }

    var asset Asset
    err = json.Unmarshal(assetJSON, &asset)
    if err != nil {
        return nil, err
    }

    return &asset, nil
}

// CompareHash compares a given hash value to a stored system hash and returns
true if they are the same; false otherwise
// Works as intended
func (s *SmartContract) CompareHash(ctx contractapi.TransactionContextInterface,
system string, hash string) (bool, error) {
    assetJSON, err := ctx.GetStub().GetState(system)
```

```
    if err != nil {
        return false, fmt.Errorf("failed to read from world state: %v", err)
    }
    if assetJSON == nil {
        return false, fmt.Errorf("the asset %s does not exist", system)
    }

    var asset Asset
    err = json.Unmarshal(assetJSON, &asset)
    if err != nil {
        return false, err
    }

    if asset.Hash == hash {
        return true, nil
    }
    return false, nil
}

// UpdateAsset issues a new asset to the world state with given details.
// Works as intended
func (s *SmartContract) UpdateAsset(ctx contractapi.TransactionContextInterface,
system string, newHash string, compliant bool, owner string, version int,
updatebaselinerequired bool) error {
    assetJSON, err := ctx.GetStub().GetState(system)
    if err != nil {
        return fmt.Errorf("failed to read from world state: %v", err)
    }
    if assetJSON == nil {
        return fmt.Errorf("the asset %s does not exist", system)
    }

    var asset Asset
    err = json.Unmarshal(assetJSON, &asset)
    if err != nil {
        return err
    }

    // Check if the hash has changed
    if asset.Hash != newHash {
        // Emit "HashMismatch" event with the asset details
        assetDetails, err := json.Marshal(asset)
        if err != nil {
            return fmt.Errorf("failed to marshal asset: %v", err)
```

```
        }
        err = ctx.GetStub().SetEvent("HashMismatch", assetDetails)
        if err != nil {
            return fmt.Errorf("failed to set event: %v", err)
        }

        // Log the event
        log.Printf("HashMismatch event emitted for asset: %s\n", asset.System)
    }

    // Update asset fields
    asset.Hash = newHash
    asset.Compliant = compliant
    asset.Owner = owner
    asset.Version = version
    asset.UpdateBaselineRequired = updatebaselinerequired

    assetJSON, err = json.Marshal(asset)
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(system, assetJSON)
}

// UpdateHash updates an existing asset in the world state with provided
parameters.
// Works as intended
func (s *SmartContract) UpdateHash(ctx contractapi.TransactionContextInterface,
system string, hash string) error {
    exists, err := s.AssetExists(ctx, system)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("the asset %s does not exist", system)
    }

    // overwriting original asset with new asset
    asset := Asset{
        System: system,
        Hash:   hash,
    }
    assetJSON, err := json.Marshal(asset)
```

```
    if err != nil {
        return err
    }

    return ctx.GetStub().PutState(system, assetJSON)
}

// DeleteAsset deletes an given asset from the world state.
// Works as intended
func (s *SmartContract) DeleteAsset(ctx contractapi.TransactionContextInterface,
system string) error {
    exists, err := s.AssetExists(ctx, system)
    if err != nil {
        return err
    }
    if !exists {
        return fmt.Errorf("the asset %s does not exist", system)
    }

    return ctx.GetStub().DelState(system)
}

// AssetExists returns true when asset with given System exists in world state
// This works as intended
func (s *SmartContract) AssetExists(ctx contractapi.TransactionContextInterface,
system string) (bool, error) {
    assetJSON, err := ctx.GetStub().GetState(system)
    if err != nil {
        return false, fmt.Errorf("failed to read from world state: %v", err)
    }

    return assetJSON != nil, nil
}

// GetAllAssets returns all assets found in world state
// This works as intended
func (s *SmartContract) GetAllAssets(ctx contractapi.TransactionContextInterface)
([]*Asset, error) {
    // range query with empty string for startKey and endKey does an
    // open-ended query of all assets in the chaincode namespace.
    resultsIterator, err := ctx.GetStub().GetStateByRange("", "")
    if err != nil {
        return nil, err
    }
```

```
    defer resultsIterator.Close()

    var assets []*Asset
    for resultsIterator.HasNext() {
        queryResponse, err := resultsIterator.Next()
        if err != nil {
            return nil, err
        }

        var asset Asset
        err = json.Unmarshal(queryResponse.Value, &asset)
        if err != nil {
            return nil, err
        }
        assets = append(assets, &asset)
    }

    return assets, nil
}

// SetUpdateBaselineRequired sets the UpdateBaselineRequired flag for a given
asset.
func (s *SmartContract) SetUpdateBaselineRequired(ctx
contractapi.TransactionContextInterface, system string, updateBaselineRequired
bool) error {
    assetJSON, err := ctx.GetStub().GetState(system)
    if err != nil {
        return fmt.Errorf("failed to read from world state: %v", err)
    }
    if assetJSON == nil {
        return fmt.Errorf("the asset %s does not exist", system)
    }

    var asset Asset
    err = json.Unmarshal(assetJSON, &asset)
    if err != nil {
        return err
    }

    // Set the UpdateBaselineRequired field
    asset.UpdateBaselineRequired = updateBaselineRequired

    assetJSON, err = json.Marshal(asset)
    if err != nil {
```

```
        return err
    }

    if updateBaselineRequired {
        // Emit "UpdateBaselineRequiredSet" event with the asset details only if
updateBaselineRequired is true
        err = ctx.GetStub().SetEvent("UpdateBaselineRequiredSet", assetJSON)
        if err != nil {
            return fmt.Errorf("failed to set event: %v", err)
        }
    }

    return ctx.GetStub().PutState(system, assetJSON)
}



func main() {
    assetChaincode, err := contractapi.NewChaincode(&SmartContract{})
    if err != nil {
        log.Panicf("Error creating chaincode: %v", err)
    }

    if err := assetChaincode.Start(); err != nil {
        log.Panicf("Error starting chaincode: %v", err)
    }
}
```

HELICS Tools – asset-exists.sh

```
#!/bin/bash

# Script to check if an asset exists.
if [ -z "$1" ]
then
        echo "[!!!] Syntax is: system (string)"
        echo "[!!!] Example: ASSET"
        exit
fi
peer chaincode query -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE -c '{"function":"AssetExists","Args":["'$1'"]}' | jq -r
```

HELICS Tools – compare-hash.sh

```
#!/bin/bash

# Script to make comparing hashes easier for humans.
if [ -z "$1" ]
then
        echo "[!!!] Syntax is: system (string), hash (string)"
        echo "[!!!] Example: ASSET 1234"
        exit
fi
peer chaincode query -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE -c '{"function":"CompareHash","Args":["'$1'", "'$2'"]}' |
jq -r
```

HELICS Tools – create-asset.sh

```
#!/bin/bash

# Script to make creating an asset easier for humans.
if [ -z "$1" ]
then
        echo "[!!!] Syntax is: system (string), hash (int), compliant (boolean),
owner (string), version (int)"
        echo "[!!!] Example: ASSET 1234 true/false ORG1 1"
        exit
fi
peer chaincode invoke -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE --peerAddresses $PEER0_ORG2_ADDRESS --tlsRootCertFiles
$PEER0_ORG2_TLS_CA_FILE -c '{"function":"CreateAsset","Args":["'$1'", "'$2'",
"'$3'", "'$4'", "'$5'"]}'
```

HELICS Tools – delete-asset.sh

```
#!/bin/bash

# Script to make deleted an asset easier for humans.
if [ -z "$1" ]
then
        echo "[!!!] Syntax is: system (string)"
        echo "[!!!] Example: ASSET"
        exit
fi
peer chaincode invoke -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE --peerAddresses $PEER0_ORG2_ADDRESS --tlsRootCertFiles
$PEER0_ORG2_TLS_CA_FILE -c '{"function":"DeleteAsset","Args":["'$1'"]}'
```

HELICS Tools – get-all-assets.sh

```
#!/bin/bash

# Script to get all assets from the ledger.

peer chaincode query -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE -c '{"function":"GetAllAssets","Args":[]}' | jq -r
```

HELICS Tools – get-history.sh

```
#!/bin/bash

# Script to make getting an assets history easier for humans.
if [ -z "$1" ]
then
        echo "[!!!] Syntax is: system (string)"
        echo "[!!!] Example: ASSET"
        exit
fi
# Export variables to ensure proper environment
export ORDERER_ADDRESS=localhost:7050
export ORDERER_TLS_HOSTNAME=orderer.example.com
export ORDERER_CA_FILE=/opt/hyperledger/fabric-samples/test-
network/organizations/ordererOrganizations/example.com/orderers/orderer.example.c
om/msp/tlscacerts/tlsca.example.com-cert.pem
export CHANNEL=mychannel
export CHAINCODE_NAME=basic
export PEER0_ORG1_ADDRESS=localhost:7051
export PEER0_ORG1_TLS_CA_FILE=/opt/hyperledger/fabric-samples/test-
network/organizations/peerOrganizations/org1.example.com/peers/peer0.org1.example
.com/tls/ca.crt
export PEER0_ORG2_ADDRESS=localhost:9051
export PEER0_ORG2_TLS_CA_FILE=/opt/hyperledger/fabric-samples/test-
network/organizations/peerOrganizations/org2.example.com/peers/peer0.org2.example
.com/tls/ca.crt
peer chaincode query -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE -c '{"function":"GetAssetHistory","Args":["'$1'"]}' | jq
```

HELICS Tools – read-asset.sh

```
#!/bin/bash

# Script to make reading an asset easier for humans.
if [ -z "$1" ]
then
        echo "[!!!] Syntax is: system (string)"
        echo "[!!!] Example: ASSET"
        exit
fi
peer chaincode query -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE -c '{"function":"ReadAsset","Args":["'$1'"]}' | jq -r
```

HELICS Tools – SetOrgEnv.sh

The following pages A-28 through A-30 describe the “SetOrgEnv.sh” script.

```
#!/bin/bash
#
# SPDX-License-Identifier: Apache-2.0




# default to using Org1
ORG=${1:-Org1}

# Exit on first error, print all commands.
set -e
set -o pipefail

# Where am I?
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )/.." && pwd )"

ORDERER_CA=${DIR}/test-
network/organizations/ordererOrganizations/example.com/tlsca/tlsca.example.com-
cert.pem
PEER0_ORG1_CA=${DIR}/test-
network/organizations/peerOrganizations/org1.example.com/tlsca/tlsca.org1.example
.com-cert.pem
PEER0_ORG2_CA=${DIR}/test-
network/organizations/peerOrganizations/org2.example.com/tlsca/tlsca.org2.example
.com-cert.pem
PEER0_ORG3_CA=${DIR}/test-
network/organizations/peerOrganizations/org3.example.com/tlsca/tlsca.org3.example
.com-cert.pem


if [[ ${ORG,,} == "org1" || ${ORG,,} == "digibank" ]]; then

   CORE_PEER_LOCALMSPID=Org1MSP
   CORE_PEER_MSPCONFIGPATH=${DIR}/test-
network/organizations/peerOrganizations/org1.example.com/users/Admin@org1.example
.com/msp
   CORE_PEER_ADDRESS=localhost:7051
```

```
   CORE_PEER_TLS_ROOTCERT_FILE=${DIR}/test-
network/organizations/peerOrganizations/org1.example.com/tlsca/tlsca.org1.example
.com-cert.pem

elif [[ ${ORG,,} == "org2" || ${ORG,,} == "magnetocorp" ]]; then

   CORE_PEER_LOCALMSPID=Org2MSP
   CORE_PEER_MSPCONFIGPATH=${DIR}/test-
network/organizations/peerOrganizations/org2.example.com/users/Admin@org2.example
.com/msp
   CORE_PEER_ADDRESS=localhost:9051
   CORE_PEER_TLS_ROOTCERT_FILE=${DIR}/test-
network/organizations/peerOrganizations/org2.example.com/tlsca/tlsca.org2.example
.com-cert.pem

else
   echo "Unknown \"$ORG\", please choose Org1/Digibank or Org2/Magnetocorp"
   echo "For example to get the environment variables to set upa Org2 shell
environment run:  ./setOrgEnv.sh Org2"
   echo
   echo "This can be automated to set them as well with:"
   echo
   echo 'export $(./setOrgEnv.sh Org2 | xargs)'
   exit 1
fi

# output the variables that need to be set
echo "export CORE_PEER_TLS_ENABLED=true"
echo "export ORDERER_CA=${ORDERER_CA}"
echo "export PEER0_ORG1_CA=${PEER0_ORG1_CA}"
echo "export PEER0_ORG2_CA=${PEER0_ORG2_CA}"
echo "export PEER0_ORG3_CA=${PEER0_ORG3_CA}"

echo "export CORE_PEER_MSPCONFIGPATH=${CORE_PEER_MSPCONFIGPATH}"
echo "export CORE_PEER_ADDRESS=${CORE_PEER_ADDRESS}"
echo "export CORE_PEER_TLS_ROOTCERT_FILE=${CORE_PEER_TLS_ROOTCERT_FILE}"

echo "export CORE_PEER_LOCALMSPID=${CORE_PEER_LOCALMSPID}"
echo "export FABRIC_CFG_PATH=$PWD/../config/"
```

HELICS Tools – update-asset.sh

```
#!/bin/bash

# Script to make updating an asset easier for humans.
if [ -z "$1" ]
then
    echo "[!!!] Syntax is: system (string), hash (int), compliant (boolean),
owner (string), version (int), updatebaselinehash (bool)"
    echo "[!!!] Example: ASSET 1234 true/false ORG 1 true/false"
    exit
fi
peer chaincode invoke -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE --peerAddresses $PEER0_ORG2_ADDRESS --tlsRootCertFiles
$PEER0_ORG2_TLS_CA_FILE -c '{"function":"UpdateAsset","Args":["'$1'", "'$2'",
"'$3'", "'$4'", "'$5'", "'$6'"]}'
```

HELICS Tools – update-hash.sh

```
#!/bin/bash

# Script to update an assets hash.
if [ -z "$1" ]
then
        echo "[!!!] Syntax is: system (string), hash (string)"
        echo "[!!!] Example: ASSET 1234"
        exit
fi
peer chaincode invoke -o $ORDERER_ADDRESS --ordererTLSHostnameOverride
$ORDERER_TLS_HOSTNAME --tls --cafile $ORDERER_CA_FILE -C $CHANNEL -n
$CHAINCODE_NAME --peerAddresses $PEER0_ORG1_ADDRESS --tlsRootCertFiles
$PEER0_ORG1_TLS_CA_FILE --peerAddresses $PEER0_ORG2_ADDRESS --tlsRootCertFiles
$PEER0_ORG2_TLS_CA_FILE -c '{"function":"UpdateHash","Args":["'$1'", "'$2'"]}'
```

INITIAL DISTRIBUTION

84310	Technical Library/Archives	(1)
58440	J. Allphin	(1)
58440	A. Quintero-Quiroga	(1)
58440	D. Kwon	(1)
58450	B. Dudley	(1)

Defense Technical Information Center
Fort Belvoir, VA 22060–6218 (1)

This page is intentionally blank.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-01-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
February 2024	Final	

4. TITLE AND SUBTITLE

Hyperledger Enhanced Layered and Integrated Cyber Security (HELICS)

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

6. AUTHORS

James H. Allphin
Anthony Quintero-Quiroga
David Kwon
Barry Dudley
NIWC Pacific

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NIWC Pacific
53560 Hull Street
San Diego, CA 92152–5001

8. PERFORMING ORGANIZATION REPORT NUMBER

TD-3429

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Project Overhead funded through the Cyber Engineering & Integration Division
53560 Hull Street
San Diego, CA 92152–5001

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This technical document describes the results of our research investigating the feasibility of combining STIG Analysis tools and Hyperledger technology to augment and enhance DoD Cyber Security.

15. SUBJECT TERMS

Enterprise Shared Data Services; Capstone Design Concept for Information Superiority; Navy data policies; rapid deployment; secure data; Enterprise Data Model; User-Centered Design; Designing Microservices; Linked Data

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James H. Allphin
U	U	U	SAR	68	19B. TELEPHONE NUMBER *(Include area code)* 760-373-6765

This page is intentionally blank.

This page is intentionally blank.

Naval Information Warfare Center (NIWC) Pacific
San Diego, CA 92152-5001