The SolarWinds Attack and Farm-to-table Methods in the Development Process: Notes

Mitigating disasters through supply chain security

Felix Pojtinger

2022-01-06

Contents

Topic: The “Solarwinds” attack and farm-to-table methods in the development process - Mitigating disasters through supply-chain security

1 Sources

```
@misc{mandiant_2020,
  title     = {Highly Evasive Attacker Leverages SolarWinds Supply
      Chain to Compromise Multiple Global Victims With SUNBURST Backdoor
      },
  url       = {https://www.mandiant.com/resources/evasive-attacker-
      leverages-solarwinds-supply-chain-compromises-with-sunburst-
      backdoor},
  journal   = {Mandiant},
  publisher = {FireEye/Mandiant},
  year      = {2020},
  month     = {Dec}
}

@techreport{sherman2019risks,
  title       = {Risks in the software supply chain},
  author      = {Sherman, Mark},
  year        = {2019},
  institution = {CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH United
      States}
}

@article{article,
  author  = {Alsabbagh, Bilal and Kowalski, Stewart},
  year    = {2015},
  month   = {07},
  pages   = {30-39},
  title   = {A Socio-technical Framework for Threat Modeling a Software
      Supply Chain},
  volume  = {13},
  journal = {IEEE Security & Privacy},
  doi     = {10.1109/MSP.2015.72}
}

@inproceedings{TorresArias2019intotoPF,
  title     = {in-toto: Providing farm-to-table guarantees for bits and
      bytes},
  author    = {Santiago Torres-Arias and Hammad Afzali and Trishank
      Karthik Kuppusamy and Reza Curtmola and Justin Cappos},
  booktitle = {USENIX Security Symposium},
  year      = {2019}
}
```

2 Part 0: The SolarWinds Attack (Highly Evasive Attacker Leverages SolarWinds Supply Chain)

- Summary
  - On 13 December 2020, FireEye detected a large supply chain attack targeting SolarWinds Orion
  - The actors behind the attack (tracked as UNC2452) gained access to data and control over both private and public institutions
  - Trojanized updates were used to get access to SolarWinds Orion since as early as Spring 2020
  - Result of the attack is lateral movement and data theft
- Backdoor
  - `SolarWinds.Orion.Core.BusinessLayer.dll` is a signed component of Orion which communicates with external servers using HTTP
  - After laying dormant for about two weeks, it receives and executes commands ("jobs")
  - Network traffic is masqueraded as the Orion Improvement Program protocol (telemetry)
  - Reconnaissance is stored with legitimate data to make detection harder
  - Backdoor is sandbox-aware using blocklists so that detection using anti-malware tools and forensics are harder
- Job capabilities
  - Transfer files
  - Execute files
  - Profile the system
  - Reboot the system
  - Disable (system) services
- Method of delivery
  - Trojanized updates were digitally signed using SolarWind's private keys
  - Delivered file is a Windows Installer Patch file
  - After installation, the host process loads the malicious DLL
  - Attempts to resolve a subdomain of `avsvmcloud.com`, which will return a command and control domain through the `CNAME` record
  - Traffic to command and control domain mimics the legitimate SolarWinds API

3 Part 1: Overview (Risks in the Software Supply Chain)

- As the SolarWinds attack has shown, supply chain attacks on any step of the supply chain can lead to significant breaches
- Let's take a look at the potentials vulnerabilities in a supply chain
- Security is a lifecycle issue:
  - Mission thread
  - Threat analysis
  - Abuse cases
  - Architecture and design principles
  - Coding rules and guidelines
  - Testing, validation and verification
  - Monitoring
  - Breach awareness
- Historically, software development didn't have a supply chain
  - Software was limited in size, function and audience
  - Each organization had their own developers
  - Each organization created their own software
- Modern software development has a complex supply chain
  - Function is largely understood and can automate existing processes
  - Is large enough to not be manageable by a single organization
  - Software can be a business opportunity for external organizations
- Modern development is assembly of existing software; the supply chain gets longer
  - App server
  - HTTP server
  - XML parser
  - C libraries
  - C compiler
  - Generated parser
  - Parser generator
  - 2nd compiler
- Toolchain corruption is starting to become an issue
  - XCodeGhost, Expensive Wall, HackTask: WeChat, Angry Bird, iOBD2

- Both COTS and FLOSS can have software issues which can be hard to pinpoint (Heartbleed, Shellshock etc. - ~1 vulnerability per 10k lines of code)
- Generated or supplied data can be vulnerabilities too, but are almost never checked
  - Data from i.e. Pandas, Numpy, TensorFlow, PyTorch etc.
  - Kaggle, Data.gov, Google Images etc.
- Reducing software supply chain risk factors
  - Supplier capability: Supplier follows best practices for internal practices too
  - Product security: The delivered product is secure
  - Product distribution: Method of delivering the product is secure
  - Operational product control: The product is used securely
- Supplier security commitment evidence
  - Employees are educated about security
    - Education level is notarized
    - Non-expired education material is used
    - Credentials of instructors are known, and notarized
  - Supplier follows suitable security design practices
    - Design guidelines are documented
    - Attack patterns have been analyzed
    - Code signing is used
- Good product distribution practices
  - Understand that you inherit the consequences of bad upstream practices
  - Require good security practices by your suppliers
  - Security of delivered products must be assessed
  - Additional risk of delivered products in their context must also be assessed
  - Internal suppliers should be used as little possible
  - Build FLOSS code only with trusted compilers
- Attacks on distribution environments
  - Source code: Shadowpad, Anti-Virus Code
  - Download site: Havex/Dragonfly, KingSlayer, CCleaner
  - Patch site: NotPetya/MeDoc
- Integrating FLOSS
  - Establish a supplier: Self or a third party focusing on FLOSS (i.e. RedHat, SUSE etc.)
  - Subject to same evaluation: Supplier capability, product security, product distribution, operational product control

4 Part 2: Framework (Socio-technical Framework for Threat Modeling a Software Supply Chain)

- Now that we've analyzed the risks associated with supply chains, let's take a look at how to model its vulnerabilites from a social and technical perspective
- Software supply chains are similar to traditional supply chains
- A supply chain is created by deploying and using a product directly or reproducing it as a new product in repetition
- Traditional supply chains can have risks
  - Late product delivery
  - Counterfeits
  - Human errors
- Software supply chains have risks too, i.e. faulty code (intentional or unintentional)
- Risk management is used to counteract these known vulnerabilities
- The first step is to create a threat model of the system
- Threat models must not be too complex in order to be useful
- Current practices in supply chain security
  - Classical CIA triad: Confidentiality, integrity, availability
  - Some newer ones: Possession, authenticity, utility
  - Specific to supply chains: Confidentiality, data integrity, source authenticity, availability, non-repudiation
  - NIST publishes best practices for securing a supply chain, but not concrete recommendations
- Thread modeling requires two processes: Modeling the system and modeling threats to the system
- Software systems and supply chains cannot be viewed as purely technical or social systems, but only as a combination of the two (a "sociotechnical" system)
- ISO 27005: A threat is a potential cause of an incident that might result in harm to systems and organizations
- Threat modeling systems should express and capture as many threats as necessary
- Actual threats: Intention, capability and opportunities, potential threats might miss one
- Both types of threats must be found, as potential threats can be prevented by acting early
- Supply chain vulnerabilities can be found by analyzing the relations of the integrated elements that compose each threat in the system
- The sociotechnical framework is based on two models/systems: A dynamic model of sociotechnical changes ("sociotechnical system") and a static model ("security by consensus")

- The social subsystem contains culture (collection of values) and structure (distribution of power)
- The technical subsystem contains methods (techniques) and machines (technical artifacts)
- All four subsystems *culture, structure, methods, machines) determine the security state of the system
- If one part of the system changes, the other parts must adapt (i.e. new, younger managers)
- The security by consensus system defines a few layers: ethical, political and legal, administrative and managerial, operation, application, operating system, hardware
- Both internal and external changes (social or technical) will affect security, so measures need to be deployed systematically
- Security frameworks make it possible to analyze supply chain security using the individual layers (vulnerabilities in one layer might for example be mitigated by another)
- By looking at software supply chains as interconnected sociotechnical systems reviewers can verify that the layers across the systems are secured similarly
- Comparing the different layers of the individual supplier's sociotechnical systems allows for creating a chain of (dis-)trust, where for example the usefulness of digital signatures for the compiler binaries is negated by compiling unsigned source code or not checking the provided binaries' signatures
- Six processes need to be analyzed: Supplier sourcing, software development and testing, software packaging, product delivery through network or software product manufacturing & physical software product delivery - this can be done by analyzing the products which are being sent between the different processes
- Supplier sourcing and product delivery link companies together, which often leads to generic processes like software packaging
- FLOSS and COTS software are transmitted from supplier sourcing to software development and testing processes, including related secrets and vulnerability information
- In the delivery process, four or five elements (secrets, vulnerabilities, source package and/or binary package, user guide) are sent to the user (over the network or physically)
- The security by consensus model can be used to find social threats too; these exist due to human errors or behavior (intentional or not)
  - Suppliers can deny having sent a software product
  - Ordered software products might not arrive in time (due to i.e. QA problems)
  - Secrets of outsourced software might be disclosed by employees, i.e. hard-coded key or seed values
  - Users might make configuration mistakes, such as choosing very short key lengths, if the user guide (which is being distributed as part of the product through the supply chain) has been tampered with

- Technical threats can be analyzed using the three bottom layers of the security by consensus model (hardware, operating system, applications)
  - Software supplier's storage hardware can be compromised, which allows externals to inject source code into i.e. source repositories or packages
  - Outsourced software's source repository can be breached to gain access to hard-coded keys (i.e. API keys in CI/CD env)
  - Download sites or update systems can be compromised
- Countermeasures to social or technical threats can be both social or technical, so both options must be considered
  - When a recipient of a software product denies receiving it, a social solution would be to require a third-party notary to prove it (political and legal layer); a technical countermeasure could be to use digital signatures
  - To prevent injection of source code during software distribution, a social solution could be a third-party escrow, and a technical countermeasure would be the use of i.e. a VPN or TLS

5 Part 3: Implementation (in-toto: Providing farm-to-table guarantees for bits and bytes)

- Using this social and technical abstract, let us now take a look at a concrete implementation of a supply chain security system, in-toto
- Examples of supply chain attacks
  - Version control systems: Linux kernel, Gentoo and Google
  - Build systems: Fedora, which allowed for signing backdoored version of security packages
  - Build environment: CCleaner
  - Software updaters: Microsoft, Adobe, Google and Linux distros
  - Are now also used by nation states against foreign states and own citizens
- Current state
  - Supply chain security is limited so securing individual steps
  - Git commit signing: Controls which devs can modify what in a repo
  - Reproducible builds: Enables building the software by multiple parties and result must be the same
  - Software delivery is taken care of by many methods (i.e. APT/DNF/Flatpak repos etc.)
- Issues with the current state

  - Securing each state is useful, but it is possible to modify a step and feed it to the next one without being noticed
  - No way to verify that the correct steps were followed or that the artifacts between the individual steps were not tampered with
  - Web server compromise of Linux Mint enabled redirects of downloaded images, thus signatures, checksums etc. were rendered useless
  - Fuzzers etc. are used, but the delivered product rarely includes information about the software it was tested with
  - Solutions designed to secure individual supply chain steps cannot guarantee the security of the entire chain as a whole

- in-toto intents to enforce the integrity of the entire supply chain

  - Declaration and signing of a layout with how and by whom the steps are to be carried out
  - The involved parties record actions and create a signed statement (link metadata) for each step
  - Link metadata can be verified for each step: Executed appropriately and by the correct party as defined in the layout

- Checks

  - Requirements: I.e. no CVEs might be in the included libraries
  - A step which creates the Spanish translation can only change `.po` files, not the application source code
  - Such requirements can reduce the scope of actions an attacker can perform, even if steps in the chain are compromised
  - Key compromises in any step does not invalidate all security measures (not a “lose-one, lose-all” solution)

- Definitions

  - Supply chain: Series of steps performed in order to create and distribute a delivered product
  - Step: Operation in a chain that takes materials (source code and artifacts) and creates products (libraries, packages, binaries etc.). May be executed in parallel or on multiple hosts to test reproducibility or for speed
  - Artifacts: Materials and products
  - Byproducts: `STDOUT`, `STDERR`, return value etc.
  - Link metadata: Contains materials, products and byproducts for the step and is signed by a functionary
  - Functionary: Party who performs a step; can commit code, build software, perform QA,

  localizes docs etc. Can also be human or multiple hosts/people to enforce redundancy or consensus.
  - Project owner: Defines the rules for the steps in a supply chain and is the foundation of trust.
  - Layout: File defined by the project owner which defines which steps should be performed by which functionaries, rules for products, byproducts, materials and inspections
  - Client: Instance which cryptographically validates delivered product
  - Delivered product: Contains software, layout and link metadata
  - Inspections: Additional actions to be performed by the client on the delivered product to validate software

- Threats which in-toto tries to protect against
  - Changing an artifact between two steps (so that modified output is the input of the next step in the chain)
  - Acting as a step (i.e. as a compiler, introducing malware into compiled binaries)
  - Providing a delivered product for which some steps were not performed
  - Including outdated or vulnerable elements
  - Providing a counterfeit version of the delivered product to users
- Security goals
  - Supply chain layout integrity: All steps have been performed in order
  - Artifact flow integrity: Artifacts can't be changed in between steps
  - Step authentication: Steps can only be performed by the intended parties
  - Implementation transparency: Existing supply chains need not be changed
  - Graceful degradation of security properties: In the event of key compromise, not all security properties should be lost
- Parties
  - Project owner: Defines the layout (i.e. FLOSS maintainer)
  - Functionaries: Perform steps in the chain (i.e. build farm or human)
  - Client: Inspects, verifies and utilizes a delivered product (i.e. end user)
- Security properties with no key compromise (breach of infrastructure or comms channels, but not keys)
  - Attacker cannot modify artifacts in between two steps or delivered product as the hash would fail validation
  - Attacker cannot provide a product with missing or reordered steps because embedded steps information would detect tampering

  - Attacker cannot provide link metadata because they can't sign it

- Security properties with key compromise

  - Fake-check: Attacker fakes that step has been run (i.e. signing or tests) when it actually didn't run
  - Product modification: Attacker provides tampered artifact as input to next step
  - Unintended retention: Attacker does not remove artifacts for next step which should have been removed, i.e. exploitable debug code
  - Arbitrary supply chain control: Attacker is able to provide alternate delivered product, which creates an alternative supply chain
  - Functionary compromise: Artifact flow integrity and step authentication is violated (attacker can forge link metadata), but attack surface is limited by rules of step that the functionary can do (i.e. unintended retention would only be possible if `DELETE` rule is missing). Can be mitigated by requiring multiple parties to do the job, thus requiring the breach to happen on i.e. multiple hosts.
  - Project owner compromise: Allows redefinition of layout; layout key however is rarely used, and should be kept offline, requiring a physical breach

- User response to failed validations

  - If the user is i.e. testing out a package in an air-gapped VM, they might choose to ignore the error
  - If the user is using a production system, they will probably detect or report the validation failure
  - Actions taken are probably similar to signature validation failures today

6 Part 4: Evaluation (in-toto: Providing farm-to-table guarantees for bits and bytes)

- Finally, let's analyze the results that the in-toto maintainers provided following some initial usage
- Debian rebuilders

  - Reproducible builds are bit-by-bit reproducible, so it is possible to build a package on a separate host and get the same hash on the result
  - A `apt-transport` for in-toto is used to provide attestations of the resulting builds using link metadata
  - Allow cryptographically asserting that a Debian package has been reproducibly built by `k` out of `n` rebuilders and the Debian build farm

  - Modification of a package would require breaching at least k out of n rebuilders, which the client can verify

- Cloud native builds with Jenkins and Kubernetes
  - Cloud-native/containerized environments require high levels of automation
  - Exporters of metadata must be host- and infrastructure-agnostic
  - Lead to implementation of a Jenkins plugin and a Kubernetes admission controller which allow for the tracking of all operations in a distributed system
  - Pipelines are coordinators, workers are functionaries and submit metadata to an in-toto metadata store
  - Admission controller validates information in the metadata store before deploying
- E2E verification of Python packages
  - Tag step outputs Python code and YAML config files as products, which is signed using a YubiKey
  - Builder receives source code from the first step and builds a Python wheel and updated metadata
  - Signer receives source code from the first step and signs it, which is separate from the builder step as the builder step can, due to Python's design, execute arbitrary code
  - Inspection ensures that the build wheel matches the materials of the signer and the original source code from the first step (end-to-end verification)
  - The Update Framework (TUF) is used to provide a higher layer of signed metadata and replay protection
  - Offline keys (in safe deposit boxes) are used, so that breaches of infrastructure don't lead to key compromise
- Storage overhead
  - In the case of the Datadog deployment, ~19% of repo size was taken up by in-toto data, which is about four times higher than TUF
  - Size is primarily larger because the data is stored as part of the pipeline
  - Large signatures also add to the size
- Network overhead
  - This overhead is of importance as large downloads increase i.e. installation times
  - Size scales with number of files, not file size due to links making up much of the size
  - Metadata size is ~44% of the package size, but can escalate if many small files are included
- Verification overhead is fairly low, with a verification overhead of ~0.6 seconds on an i7-6500U with 8 GB of RAM

- Protection against previous breaches
  - Three categories of breaches: Control of infrastructure but not functionary keys, control of part of infrastructure or keys of specific functionary, and control of the entire supply chain by compromising project owner infrastructure including keys
  - The majority of surveyed attacks (23/30) did not include a key compromise, where in-toto's client inspection would have detected the tampering
  - In the Keydnap attack, where an Apple developer certificate was stolen and used to sign a malicious software package, inspection with in-toto would have detected the attack due to a unauthorized functionary signing the link metadata
  - In another attack, the developer's SSH key was used to sign a malicious Python package, which could have been prevented with in-toto as the files extracted from the malicious package would not match the source from the first step in the chain
  - CCleaner and RedHat attacks would not have been effective against Reproducible Builds and Datadog's setup due to the threshold mechanism (where multiple hosts build the artifacts), but the Cloud-Native deployment would not detect these attacks
  - In total, both Cloud-Native (with 83% prevention as a result of in-toto usage) and reproducible builds (with 90% prevention) integrations of in-toto would prevent most historical attacks
  - Integration of in-toto with secure update systems like Datadog's deployment provides further protection (against 100% of surveyed historical supply chain attacks in this case)