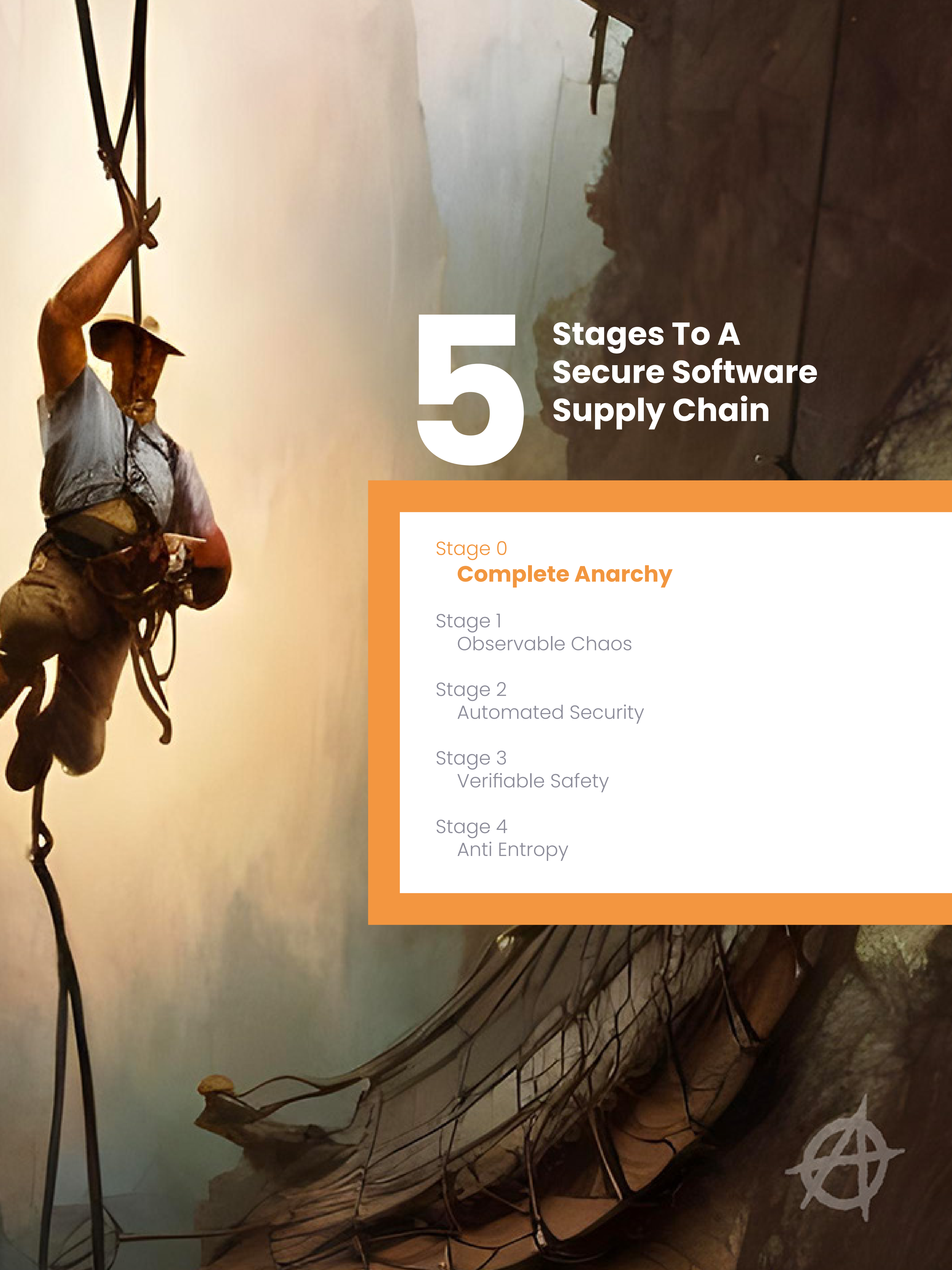
activestate

The Journey To Software Supply Chain Security

Dana Crane
with **Scott Robertson**



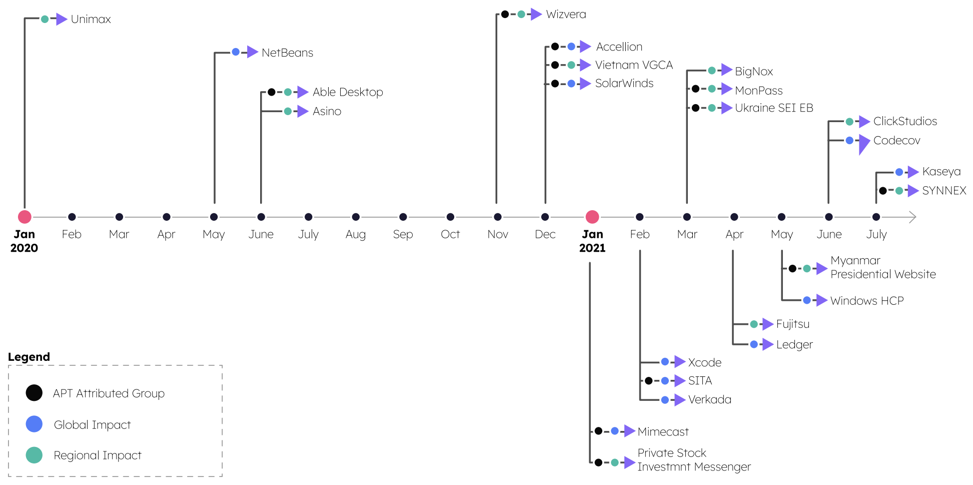
5 Stages To A Secure Software Supply Chain

Introduction

HIstorically, hackers have searched for needle-sized exploits in a worldwide haystack of corporations. All that changed in 2020 with the SolarWinds hack, which showed that one compromised development environment at a key software vendor can result in trojanized patches and software updates being propagated downstream to tens of thousands of customers.

In other words, bad actors have discovered economies of scale: a single cyberattack against a popular software vendor can grant hackers access to corporations and governments around the globe in today's internet-connected world.

Copycat attacks quickly followed:



Source: ENISA Threat Landscape for Supply Chain Attacks

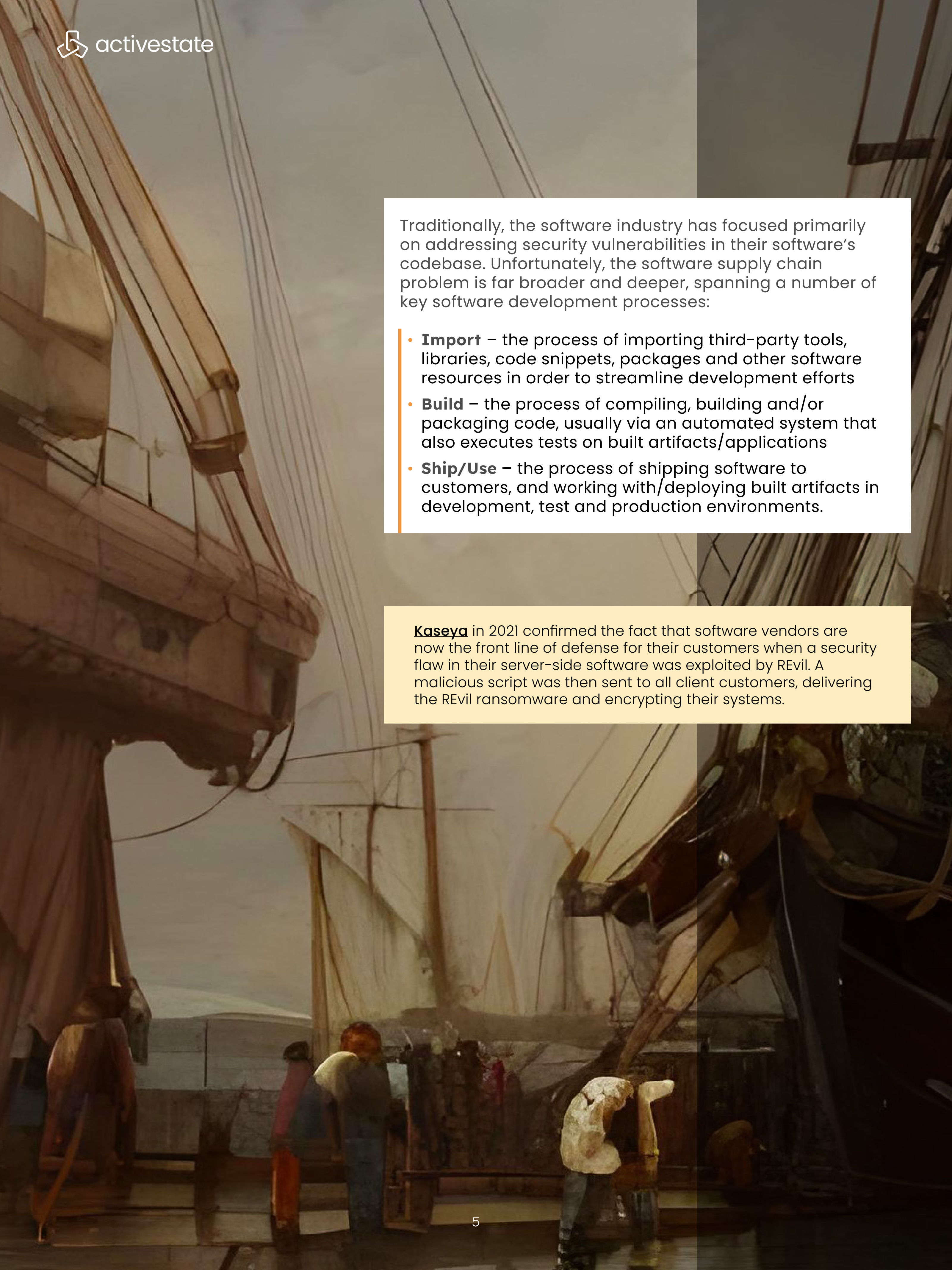
SolarWinds became the poster child for software supply chain attacks when hackers inserted a malicious DLL into their software build process prior to the signing step. Signing is widely considered a best practice to ensure that code has not been altered or corrupted since the application was signed.

But the real value of signing to most customers is the establishment of trust. This is why the SolarWinds hack was especially pernicious: it effectively undermined trust in signed software.

“*There has been an astonishing 742% average annual increase in Software Supply Chain attacks over the past 3 years.*”

State of the Software Supply Chain[1]

https://www.sonatype.com/state-of-the-software-supply-chain/introduction



Traditionally, the software industry has focused primarily on addressing security vulnerabilities in their software's codebase. Unfortunately, the software supply chain problem is far broader and deeper, spanning a number of key software development processes:

- **Import** – the process of importing third-party tools, libraries, code snippets, packages and other software resources in order to streamline development efforts
- **Build** – the process of compiling, building and/or packaging code, usually via an automated system that also executes tests on built artifacts/applications
- **Ship/Use** – the process of shipping software to customers, and working with/deploying built artifacts in development, test and production environments.

Kaseya in 2021 confirmed the fact that software vendors are now the front line of defense for their customers when a security flaw in their server-side software was exploited by REvil. A malicious script was then sent to all client customers, delivering the REvil ransomware and encrypting their systems.

The problem is compounded because the breadth and depth of the software supply chain affords multiple points of entry for malicious actors who are always looking for the weakest link in the chain to exploit:

- **Breadth** – most organizations work with multiple open source languages, and import their code from more than one public repository. Because there are no industry-wide standards in place today, each language and repository must be treated uniquely.
- **Depth** – There is a large set of best-practice application and system security & integrity controls that can help, but only the largest enterprises can hope to implement and maintain them all.
- **Change** – no supply chain is ever set in stone: open source authors change; packages are constantly updated, become vulnerable, and get patched. Languages go EOL, repositories move, trusted vendors change, etc, making it difficult to keep up.

Security has always been seen as a blocker to getting software to market, and with the exception of security-conscious industries, is typically relegated to a back seat in the software development process. In other words, the threat to revenue is often seen as greater than any potential security threat.

As a result, the US government has taken the unprecedented step of effectively legislating software supply chain security, which will force US government software suppliers to comply with a set of supply chain security requirements by June 11, 2023. Detailed in the government's Executive Order 14028 focused on "improving the nation's cybersecurity", these requirements can be met by following the recommendations laid out in this eBook.

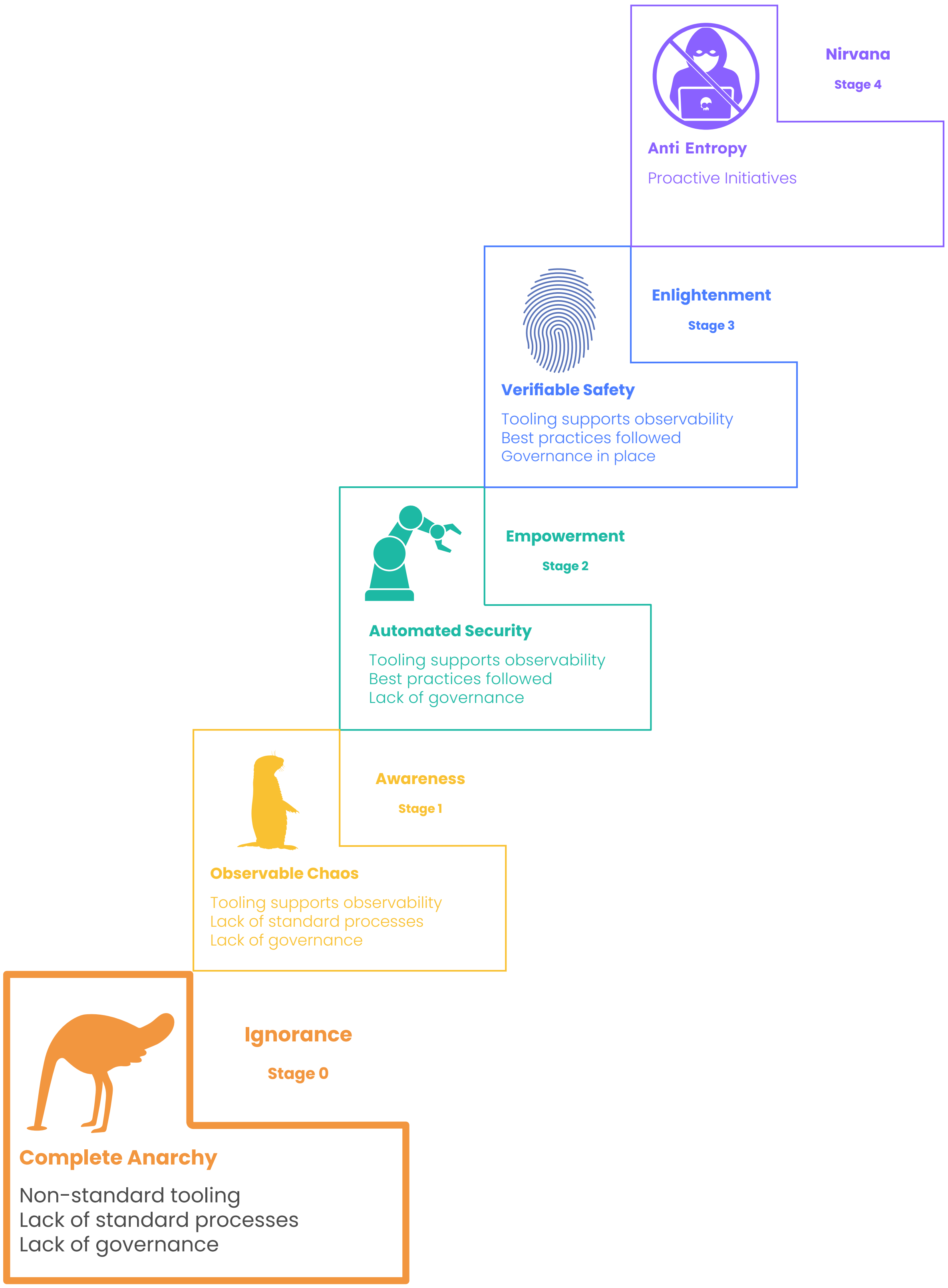
CircleCI, the popular CI/CD vendor, offers the latest proof that the industry has learned nothing from the SolarWinds incident, and continues to have an appetite for risk that far outweighs the perceived threat to their software supply chain.

It also illustrates the need to secure not just the build process, but developer desktop environments as well. This is potentially a far more challenging task given developer ingenuity at finding creative solutions to restrictive problems.

The Five Stages to Software Supply Chain Security

Every journey begins with a single step, but for many smaller organizations the first step can represent such a significant cultural change that they never commit to it. On the other hand, established enterprises are likely to be well down the path, having had best practices and supporting tooling in place for many years.

With that in mind, organizations that want to secure their supply chain can use the following journey to create a roadmap of tools, processes and initiatives, which will also allow them to comply with key US secure supply chain requirements:



Nirvana
Stage 4

Anti Entropy
Proactive Initiatives

Enlightenment
Stage 3

Verifiable Safety
Tooling supports observability
Best practices followed
Governance in place

Empowerment
Stage 2

Automated Security
Tooling supports observability
Best practices followed
Lack of governance

Awareness
Stage 1

Observable Chaos
Tooling supports observability
Lack of standard processes
Lack of governance

Ignorance
Stage 0

Complete Anarchy
Non-standard tooling
Lack of standard processes
Lack of governance

"Our current practice of implicitly trusting the packages we get from public repositories is no better or worse than the rest of the software industry."

32% of ActiveState Supply Chain Security Survey Respondents Agree

https://www.activestate.com/resources/datasheets/software-supply-chain-security-survey-report/

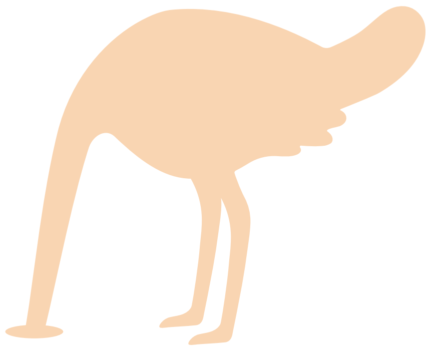
Ignorance
Stage 0
Complete Anarchy

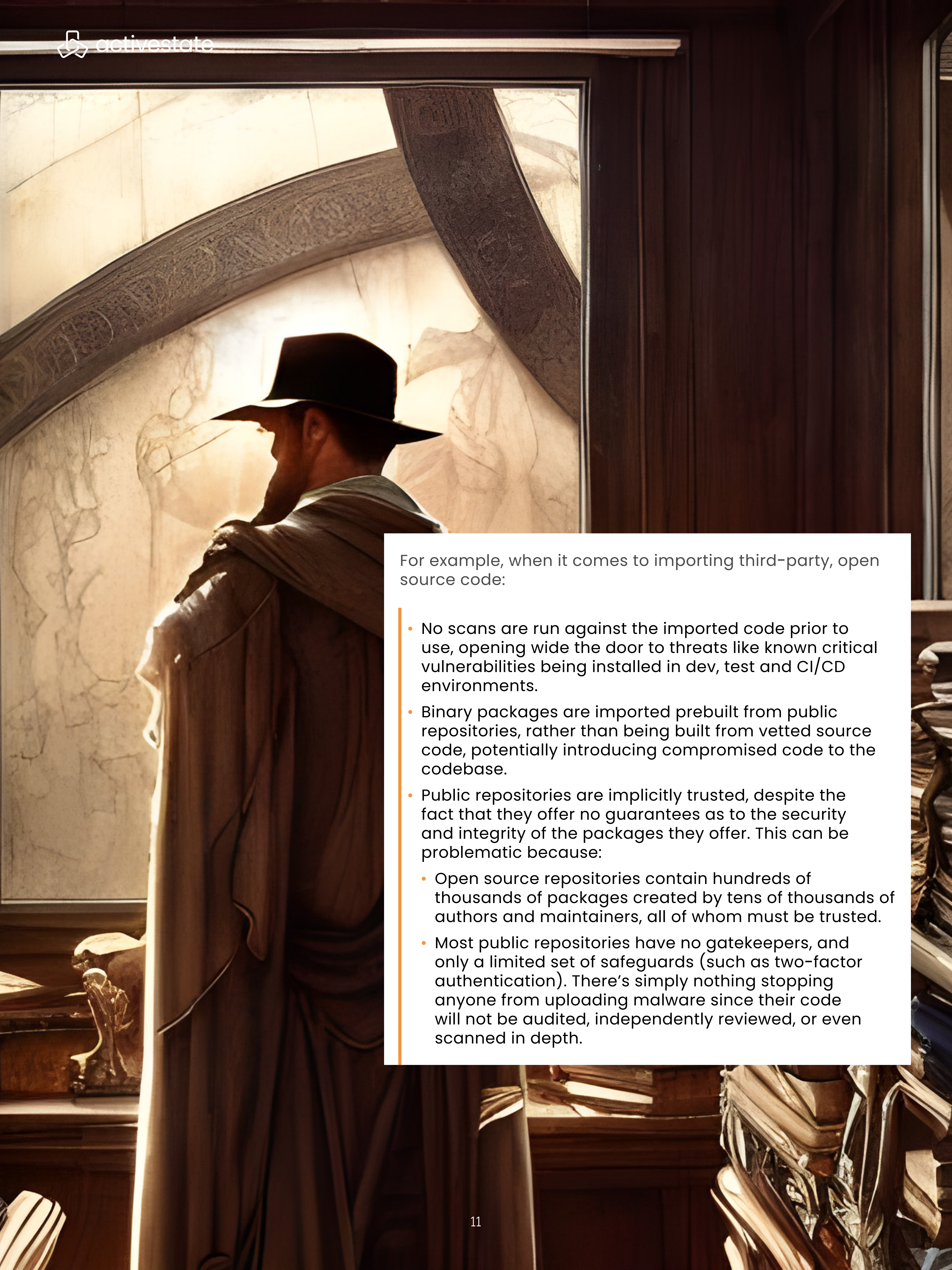
Most organizations would hesitate to characterize the way they work as "complete anarchy," but anarchy is really a double-edged sword, allowing organizations to exist on the basis of voluntary cooperation despite the disorder due to a lack of controlling systems.

Typical characteristics include:

- **Non-Standard Tooling -** while there is agreement on shared tooling (such as the code repository, for example), every developer has their own set of preferred desktop tools.
- **Lack of Standard Processes -** code may or may not be peer reviewed; warnings are investigated (or not) irrespective of severity; libraries are updated (or not) depending on local need, and so on.
- **Lack of Governance -** with no standards to apply, it's pointless to introduce a governance layer to ensure processes are followed.

While the above description may seem daunting, in practice, this way of working can actually be quite empowering, liberating everyone to be as creative as possible. But it also means that security is an afterthought, if it's thought of at all.





For example, when it comes to importing third-party, open source code:

- No scans are run against the imported code prior to use, opening wide the door to threats like known critical vulnerabilities being installed in dev, test and CI/CD environments.
- Binary packages are imported prebuilt from public repositories, rather than being built from vetted source code, potentially introducing compromised code to the codebase.
- Public repositories are implicitly trusted, despite the fact that they offer no guarantees as to the security and integrity of the packages they offer. This can be problematic because:
  - Open source repositories contain hundreds of thousands of packages created by tens of thousands of authors and maintainers, all of whom must be trusted.
  - Most public repositories have no gatekeepers, and only a limited set of safeguards (such as two-factor authentication). There's simply nothing stopping anyone from uploading malware since their code will not be audited, independently reviewed, or even scanned in depth.

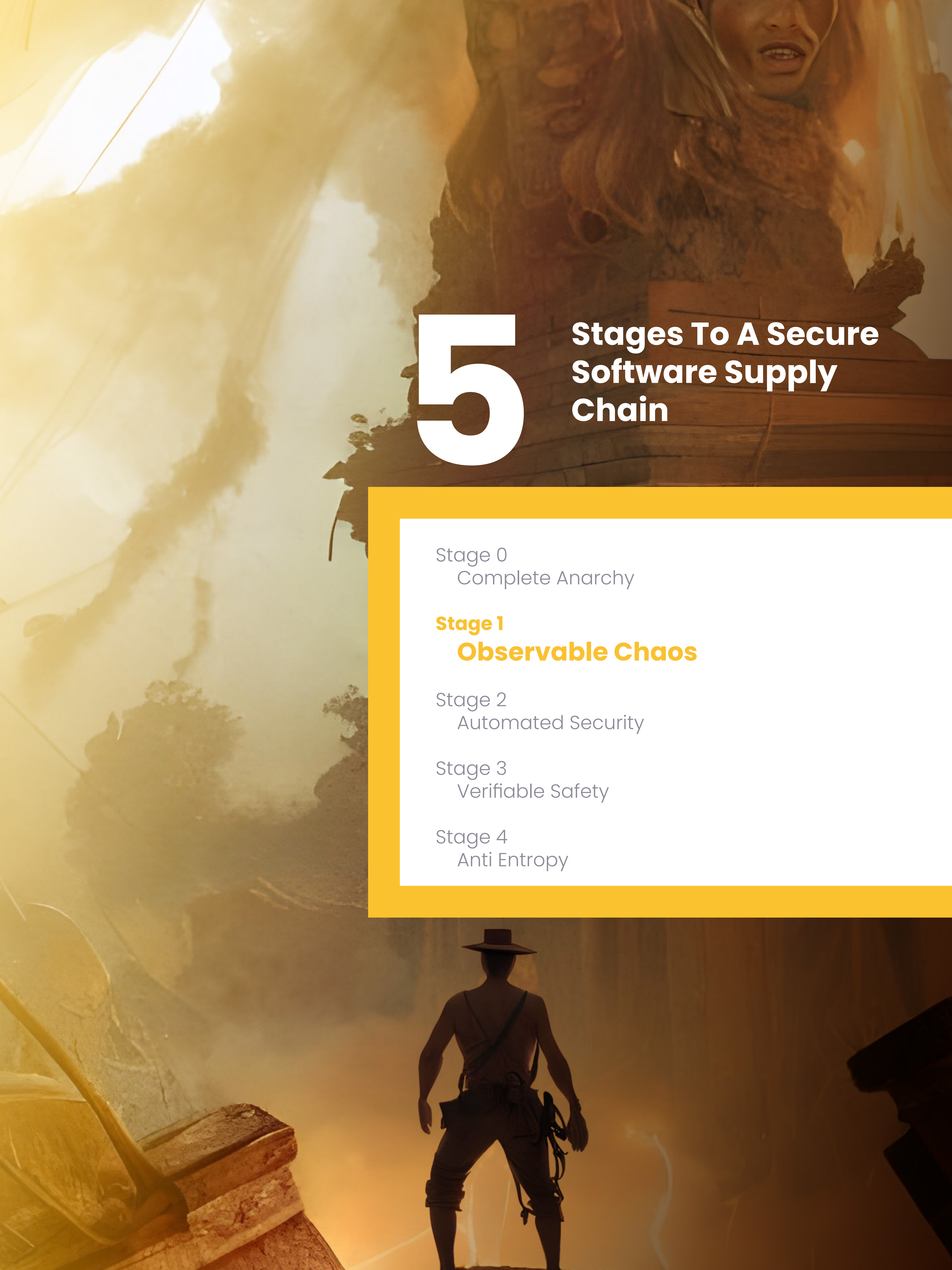
This reliance on prebuilt components typically means the only artifacts being generated by a build process are versions of the organization's application/service. But a lack of standard processes and security safeguards can mean:

- Builds are created in a non-reproducible way, making it very difficult to verify issues from one build to the next.
- Build scripts can be modified at any point, providing a foothold for bad actors to compromise them and exploit the build system.
- Build environments for each step in the process are reused, increasing the chance they become corrupted or compromised.
- The build system is connected to the Internet, potentially allowing dynamic packages to include remote, unexpected resources.
- Artifacts generated by the build process are unsigned, meaning there is no way to verify whether they have been compromised between the time they were built and the time they're deployed

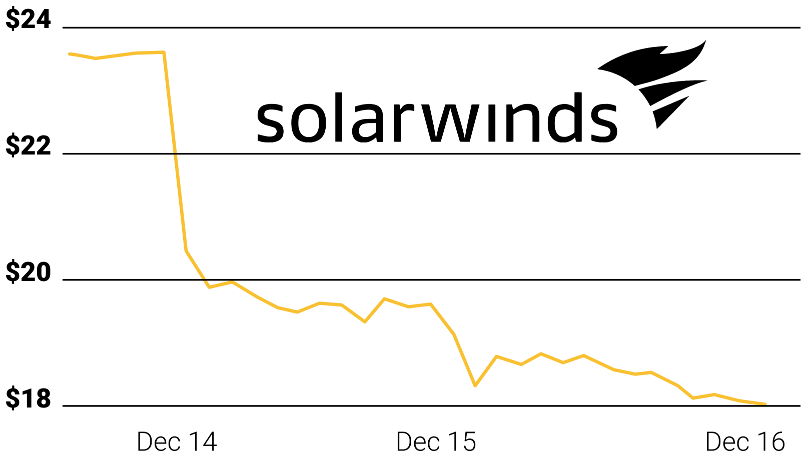
If all this sounds like your organization, you most likely work at a startup. But it can also be characteristic of open source or ad hoc projects – anywhere that developers gather to collaborate without the friction of process-heavy software development.

Unfortunately, moving from Stage 0 to Stage 1 will likely be the greatest challenge you face on your software supply chain journey since it will require a completely different culture.

After all, ignoring security is not a sign of wilful ignorance, but rather an expedient designed to maximize code output. As such, it may not be possible to implement Stage 1 until you've released your product and established a market for it.

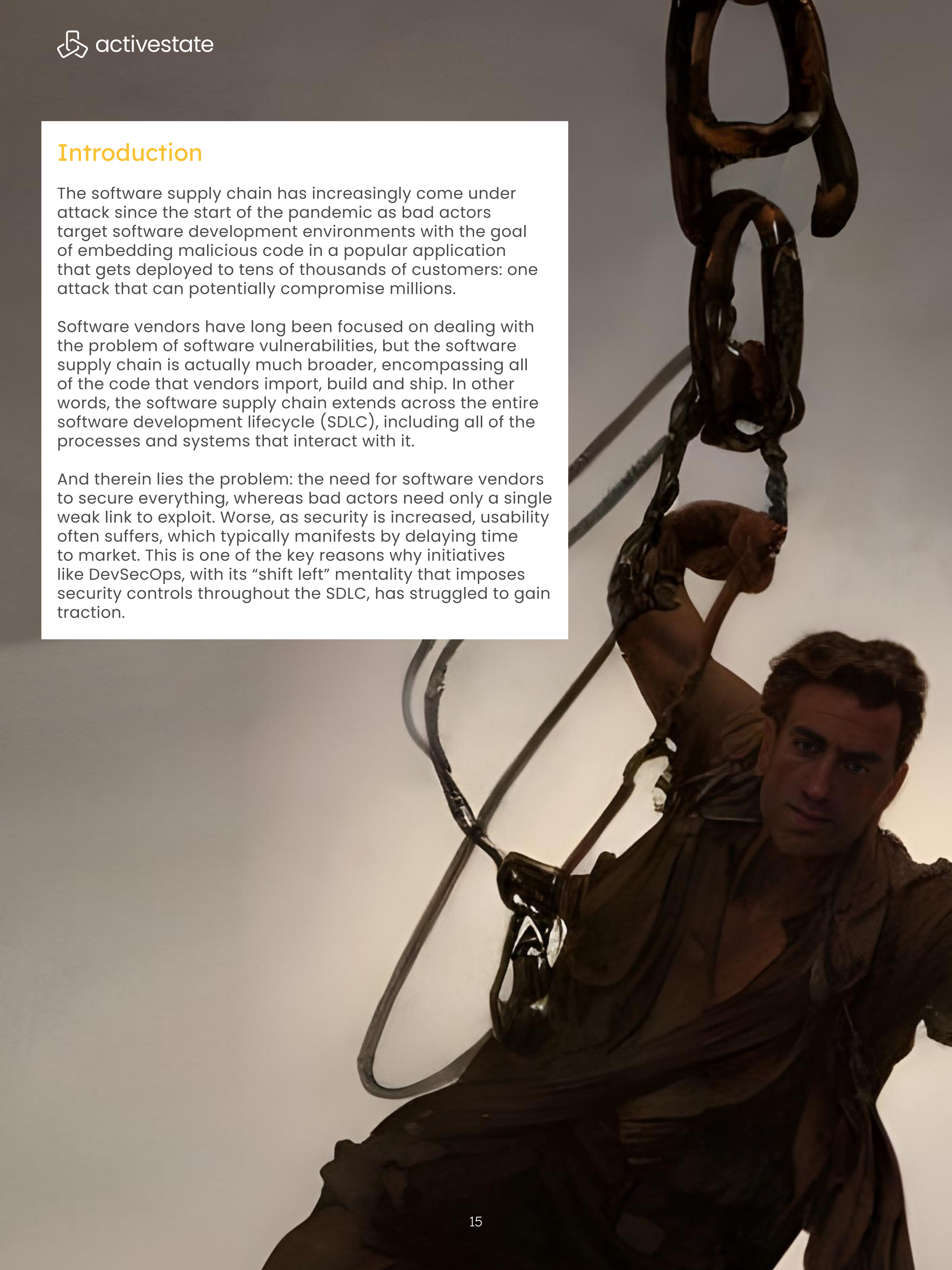


5 Stages To A Secure Software Supply Chain



Solarwinds hack impacted:

- 80% of the Fortune 500
- Top 10 US telecoms
- Top 5 US accounting firms
- CISA, FBI, NSA & all 5 branches of the US military



Introduction

The software supply chain has increasingly come under attack since the start of the pandemic as bad actors target software development environments with the goal of embedding malicious code in a popular application that gets deployed to tens of thousands of customers: one attack that can potentially compromise millions.

Software vendors have long been focused on dealing with the problem of software vulnerabilities, but the software supply chain is actually much broader, encompassing all of the code that vendors import, build and ship. In other words, the software supply chain extends across the entire software development lifecycle (SDLC), including all of the processes and systems that interact with it.

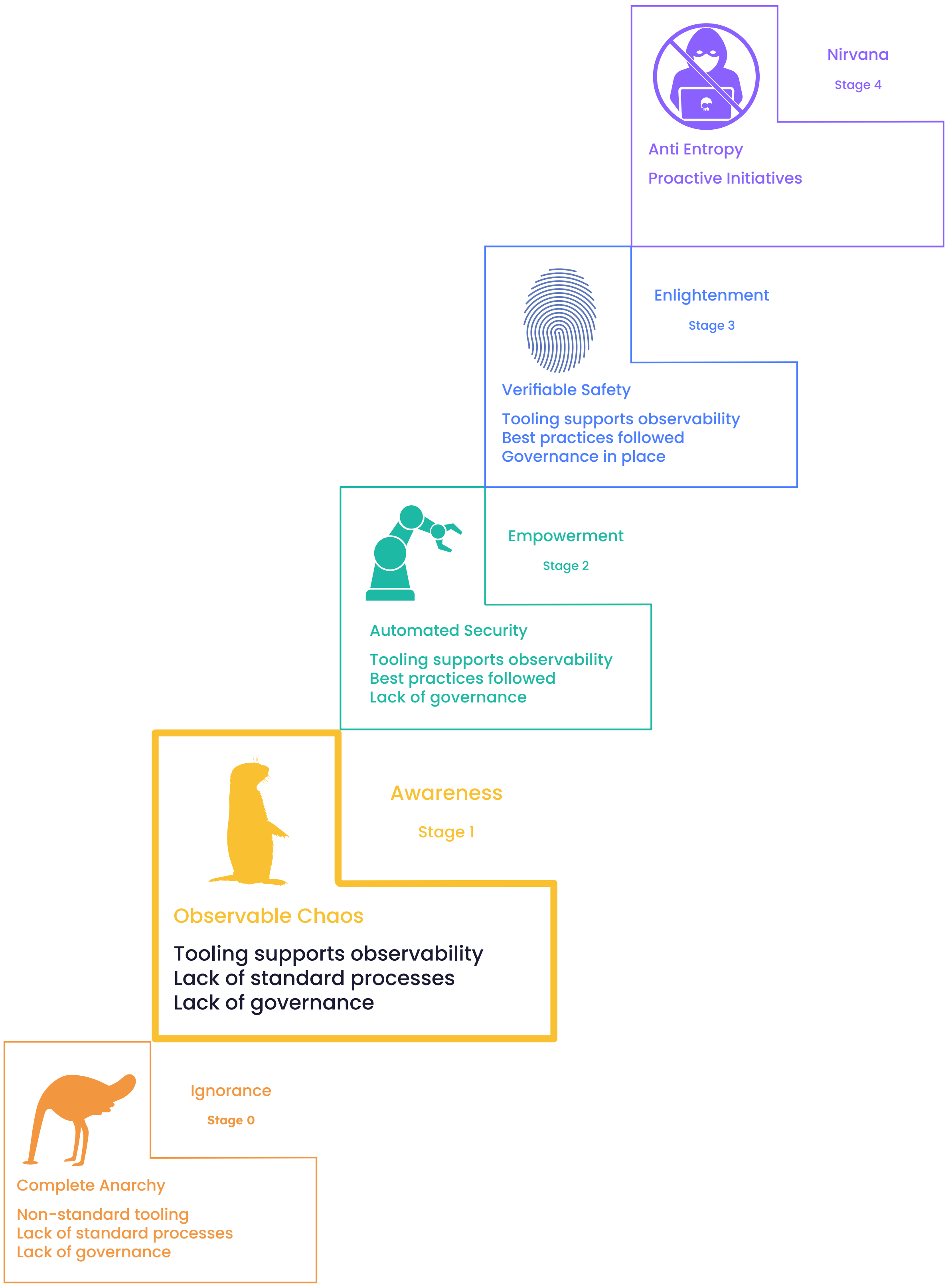
And therein lies the problem: the need for software vendors to secure everything, whereas bad actors need only a single weak link to exploit. Worse, as security is increased, usability often suffers, which typically manifests by delaying time to market. This is one of the key reasons why initiatives like DevSecOps, with its “shift left” mentality that imposes security controls throughout the SDLC, has struggled to gain traction.

In response to the growing threat, as well as the reluctance of software vendors to embrace a security-first mindset, the US government has taken the exceptional step of imposing supply chain security requirements. Effective from June 2023, any vendor of software deployed at (or even coming in contact with systems at) US government agencies or departments must comply or risk losing their contract. While the guidelines are extensive, key requirements for software vendors include:

- **SBOMs –** vendors must provide a machine-readable list of all the components that make up their software application, including third party libraries and integrations.
- **Secure Software Development –** vendors must adopt secure software development best practices, starting with detecting and resolving security vulnerabilities.

In much the same way that European Union (EU) General Data Protection Regulation (GDPR) requirements were adopted worldwide for fear of losing out on EU revenue, the US' secure supply chain requirements are likely to become just as widespread.

With that in mind, this chapter focuses on Stage 1 (Observable Chaos) of the Secure Supply Chain Journey, which can help organizations get started on the path to securing their software supply chain and complying with US requirements:


Nirvana
Stage 4
Anti Entropy
Proactive Initiatives
Enlightenment
Stage 3
Verifiable Safety
Tooling supports observability
Best practices followed
Governance in place
Empowerment
Stage 2
Automated Security
Tooling supports observability
Best practices followed
Lack of governance
Awareness
Stage 1
Observable Chaos
Tooling supports observability
Lack of standard processes
Lack of governance
Ignorance
Stage 0
Complete Anarchy
Non-standard tooling
Lack of standard processes
Lack of governance

“

Observability means transparency for all stakeholders who need to know what’s in the software at a granular level, and trust that what they are receiving is secure.

”

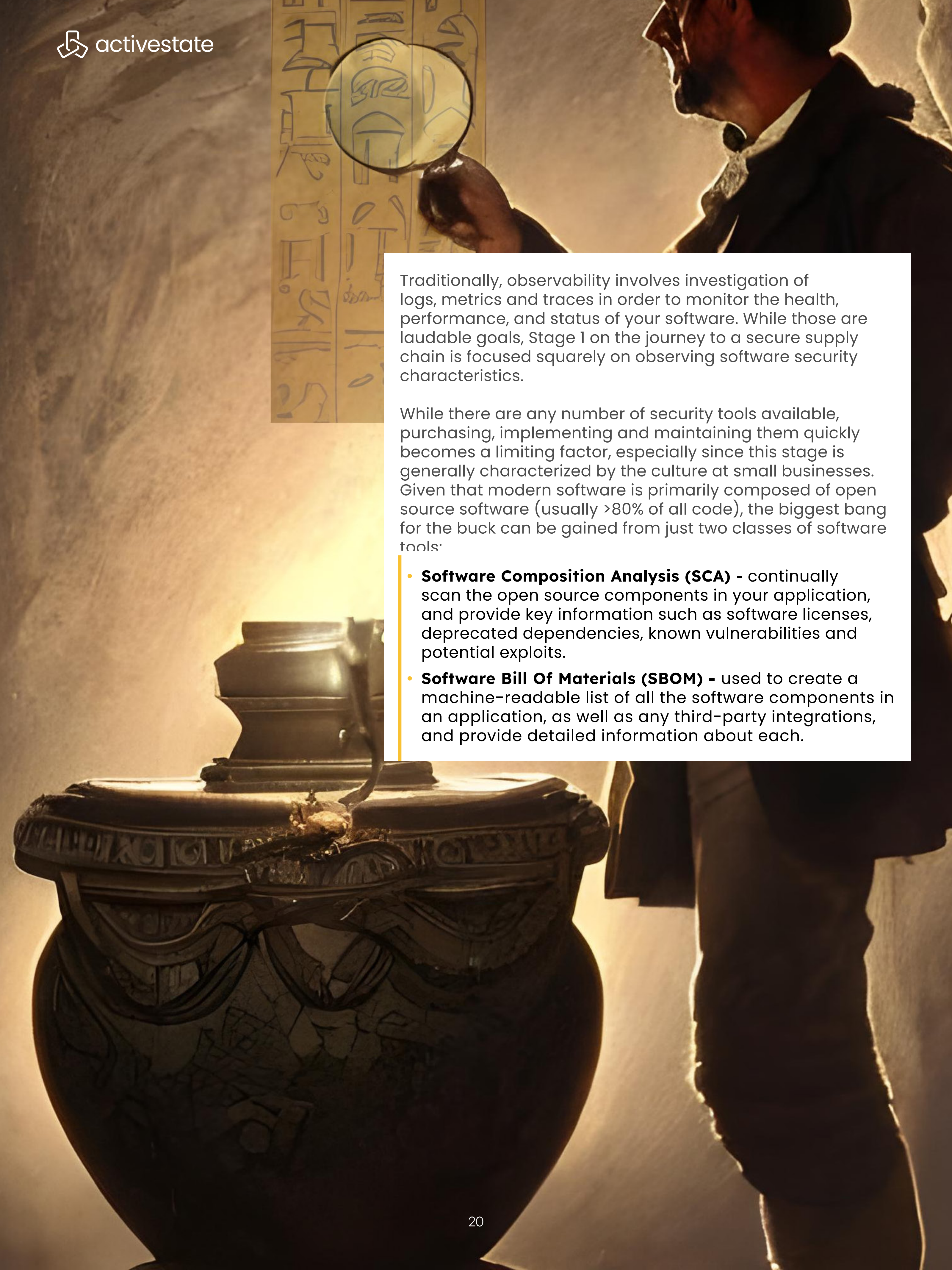
Customer quote from one of the world’s largest ISVs

Awareness
Stage 1
Observable Chaos

The first real step on the journey to a secure software supply chain is determining the security status of your existing codebase, and observing how that status evolves over time. While you may have left behind the "complete anarchy" described in Stage 0, a lack of processes and governance mean Stage 1 can still feel chaotic as you get your arms around the problem.

Characteristics include:

- **Standard Tooling -** a set of common software tools that provide insight into key security measures, such as vulnerabilities, unsecure code, malicious packages, etc.
- **Lack of Standard Processes -** vulnerabilities are remediated (or not) irrespective of criticality; security warnings are investigated (or not) irrespective of severity, and so on.
- **Lack of Governance -** with no standards to apply, it's pointless to introduce a governance layer to ensure processes are followed.



Traditionally, observability involves investigation of logs, metrics and traces in order to monitor the health, performance, and status of your software. While those are laudable goals, Stage 1 on the journey to a secure supply chain is focused squarely on observing software security characteristics.

While there are any number of security tools available, purchasing, implementing and maintaining them quickly becomes a limiting factor, especially since this stage is generally characterized by the culture at small businesses. Given that modern software is primarily composed of open source software (usually >80% of all code), the biggest bang for the buck can be gained from just two classes of software tools:

- **Software Composition Analysis (SCA) -** continually scan the open source components in your application, and provide key information such as software licenses, deprecated dependencies, known vulnerabilities and potential exploits.
- **Software Bill Of Materials (SBOM) -** used to create a machine-readable list of all the software components in an application, as well as any third-party integrations, and provide detailed information about each.

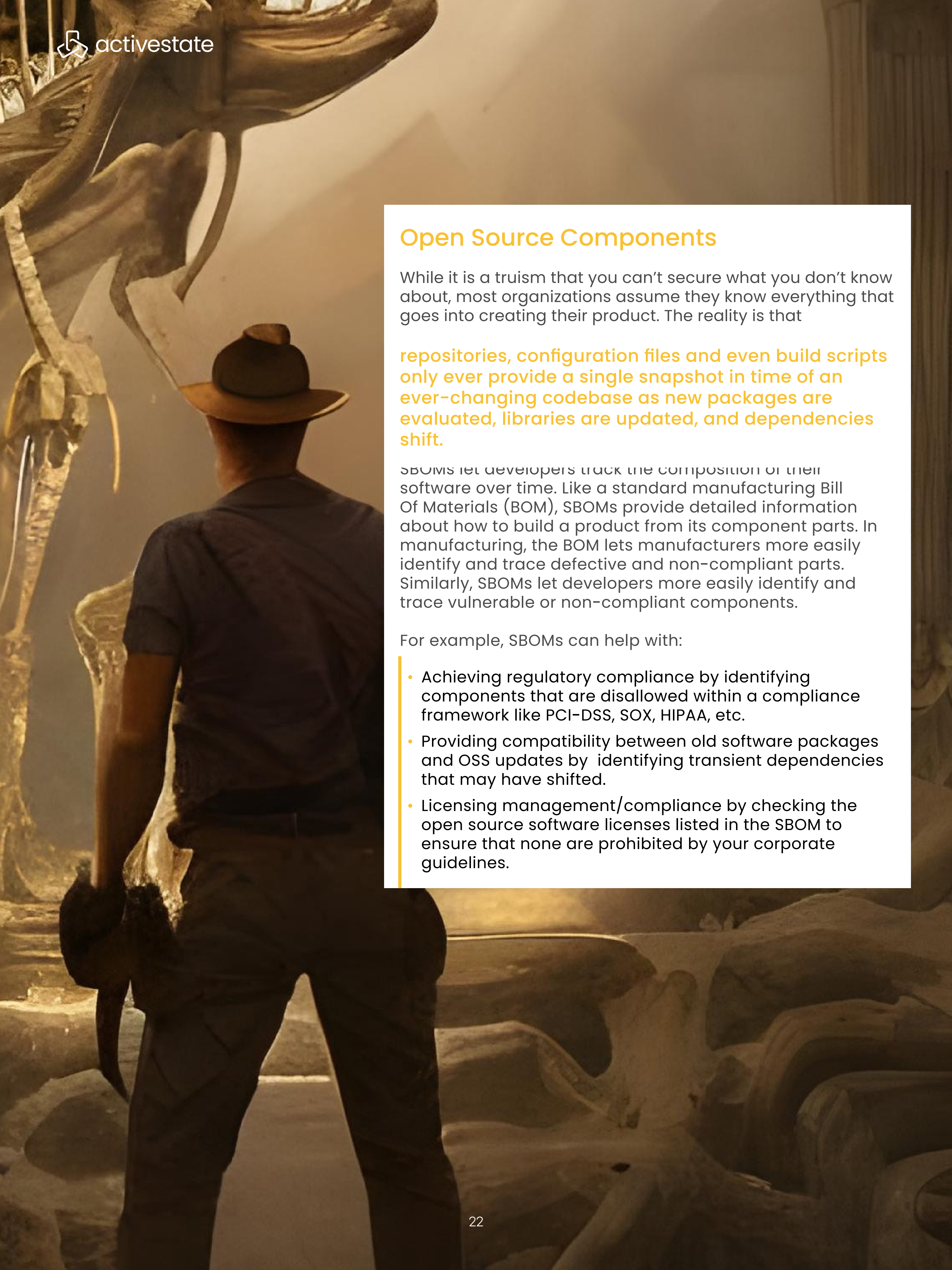
Software Vulnerabilities

The lowest hanging fruit, and where most organizations begin securing their software supply chain, is by addressing Common Vulnerabilities and Exposures (CVEs) in their open source software. While SCA tools will automatically detect a vulnerability in your codebase and notify you about it, decreasing Mean Time To Detection (MTTD), remediating vulnerabilities typically requires a lengthy process:

- **Investigate –** depending on how a component has been implemented, an application may or may not be subject to a vulnerability. Developers need to dedicate time to determine the impact.
- **Rebuild –** if a patch or upgrade is applied to a component, there's always the chance that the update will break the build. Complications can also arise when upgrades result in conflicts with other components, leading to dependency hell.
- **Retest –** this task often includes manual testing in addition to automated testing.
- **Redeploy –** organizations often need to schedule a deployment time, or even wait for the next deployment window if their production system is locked down.

All of which is why Mean Time To Remediation (MTTR) can range from 60 to 150 days. And that's the best case scenario, since multiple reports consistently confirm that the vast majority of codebases are never updated unless a critical vulnerability of note (such as Heartbleed, Log4j, etc) forces a revision.

Small businesses with limited resources may want to use a service like Github's Dependabot that not only notifies you of vulnerabilities in your GIthub repository, but can also help you automatically pull in an updated version of the vulnerable component. ActiveState goes one step further, automatically rebuilding your runtime environment with the updated component, so you need only retest and redeploy your application.



Open Source Components

While it is a truism that you can't secure what you don't know about, most organizations assume they know everything that goes into creating their product. The reality is that

repositories, configuration files and even build scripts only ever provide a single snapshot in time of an ever-changing codebase as new packages are evaluated, libraries are updated, and dependencies shift.

SBOMs let developers track the composition of their software over time. Like a standard manufacturing Bill Of Materials (BOM), SBOMs provide detailed information about how to build a product from its component parts. In manufacturing, the BOM lets manufacturers more easily identify and trace defective and non-compliant parts. Similarly, SBOMs let developers more easily identify and trace vulnerable or non-compliant components.

For example, SBOMs can help with:

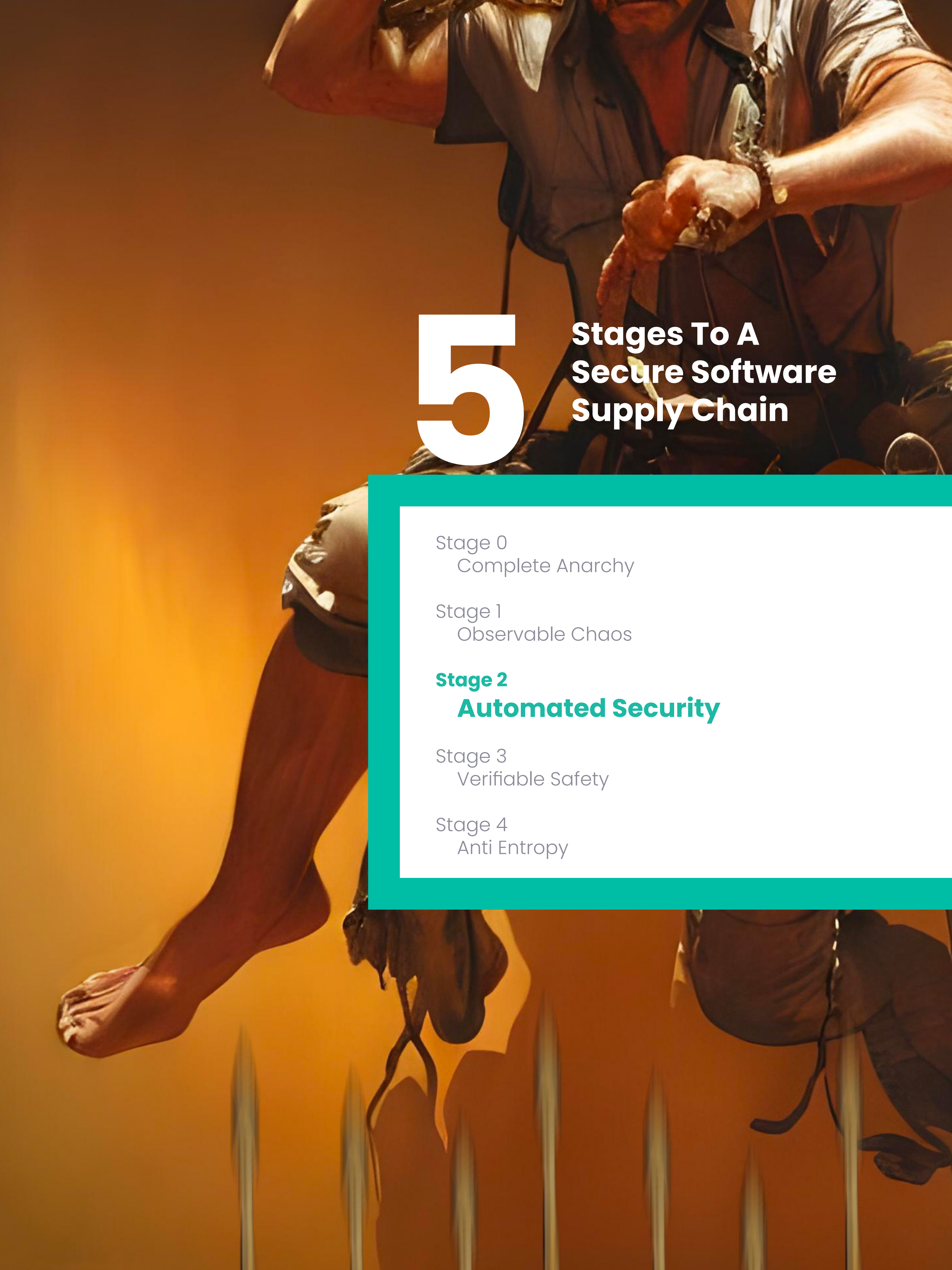
- Achieving regulatory compliance by identifying components that are disallowed within a compliance framework like PCI-DSS, SOX, HIPAA, etc.
- Providing compatibility between old software packages and OSS updates by identifying transient dependencies that may have shifted.
- Licensing management/compliance by checking the open source software licenses listed in the SBOM to ensure that none are prohibited by your corporate guidelines.

For buyers, such as US government departments and agencies, SBOMs allow them to easily identify the impact of vulnerabilities across all the applications they deploy. Fortunately, most SCA tools now include SBOM generation, but many development platforms also offer plug-n-play SBOM generation, including:

- Microsoft's SPDX sbom-tool
- GitLab's CycloneDX generator
- Anchore's SBOM GitHub Action
- ActiveState's SPDX SBOM

SCA, SBOM and vulnerability remediation tools are a good starting point, but without a framework to ensure that people, skills, and tools are being used in a consistent manner, it's just that: a start.

The next stage in the journey to software supply chain security focuses on the import, build and deployment processes you need to ensure that the tools are used effectively and consistently.



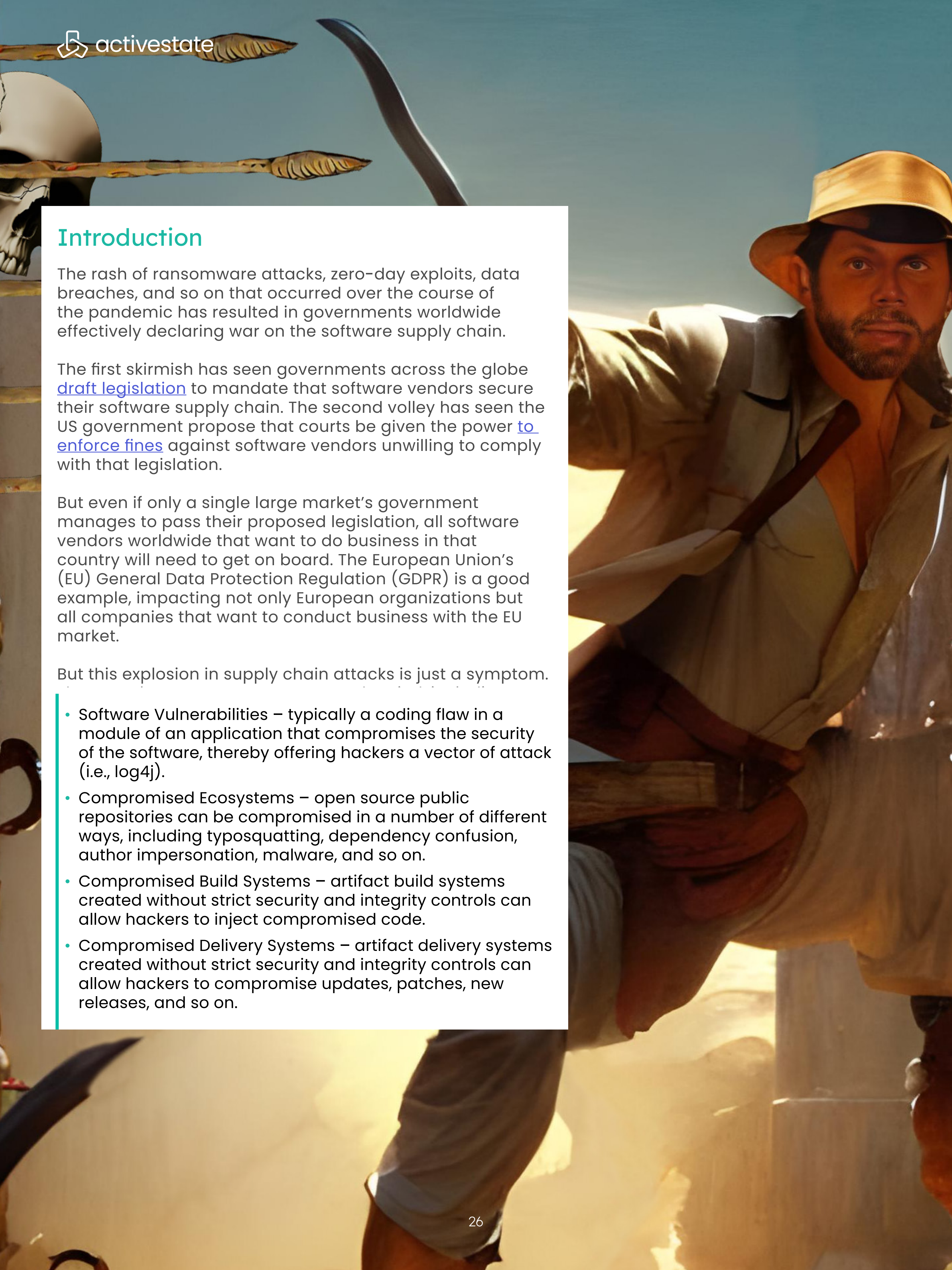
5 Stages To A Secure Software Supply Chain

“

We were surprised to find that one of the biggest players in the Python market still uses manual processes to build their distro. Our security team wouldn’t let us use them.

”

– US Armed Forces customer



Introduction

The rash of ransomware attacks, zero-day exploits, data breaches, and so on that occurred over the course of the pandemic has resulted in governments worldwide effectively declaring war on the software supply chain.

The first skirmish has seen governments across the globe draft legislation to mandate that software vendors secure their software supply chain. The second volley has seen the US government propose that courts be given the power to enforce fines against software vendors unwilling to comply with that legislation.

But even if only a single large market's government manages to pass their proposed legislation, all software vendors worldwide that want to do business in that country will need to get on board. The European Union's (EU) General Data Protection Regulation (GDPR) is a good example, impacting not only European organizations but all companies that want to conduct business with the EU market.

But this explosion in supply chain attacks is just a symptom.

- Software Vulnerabilities – typically a coding flaw in a module of an application that compromises the security of the software, thereby offering hackers a vector of attack (i.e., log4j).
- Compromised Ecosystems – open source public repositories can be compromised in a number of different ways, including typosquatting, dependency confusion, author impersonation, malware, and so on.
- Compromised Build Systems – artifact build systems created without strict security and integrity controls can allow hackers to inject compromised code.
- Compromised Delivery Systems – artifact delivery systems created without strict security and integrity controls can allow hackers to compromise updates, patches, new releases, and so on.

In other words, the software supply chain extends across the entire Software Development Lifecycle (SDLC). This is why the US government tasked the National Institute for Standards and Technology (NIST) to create a compendium of security best practices not only for developers, but suppliers and customers as well.

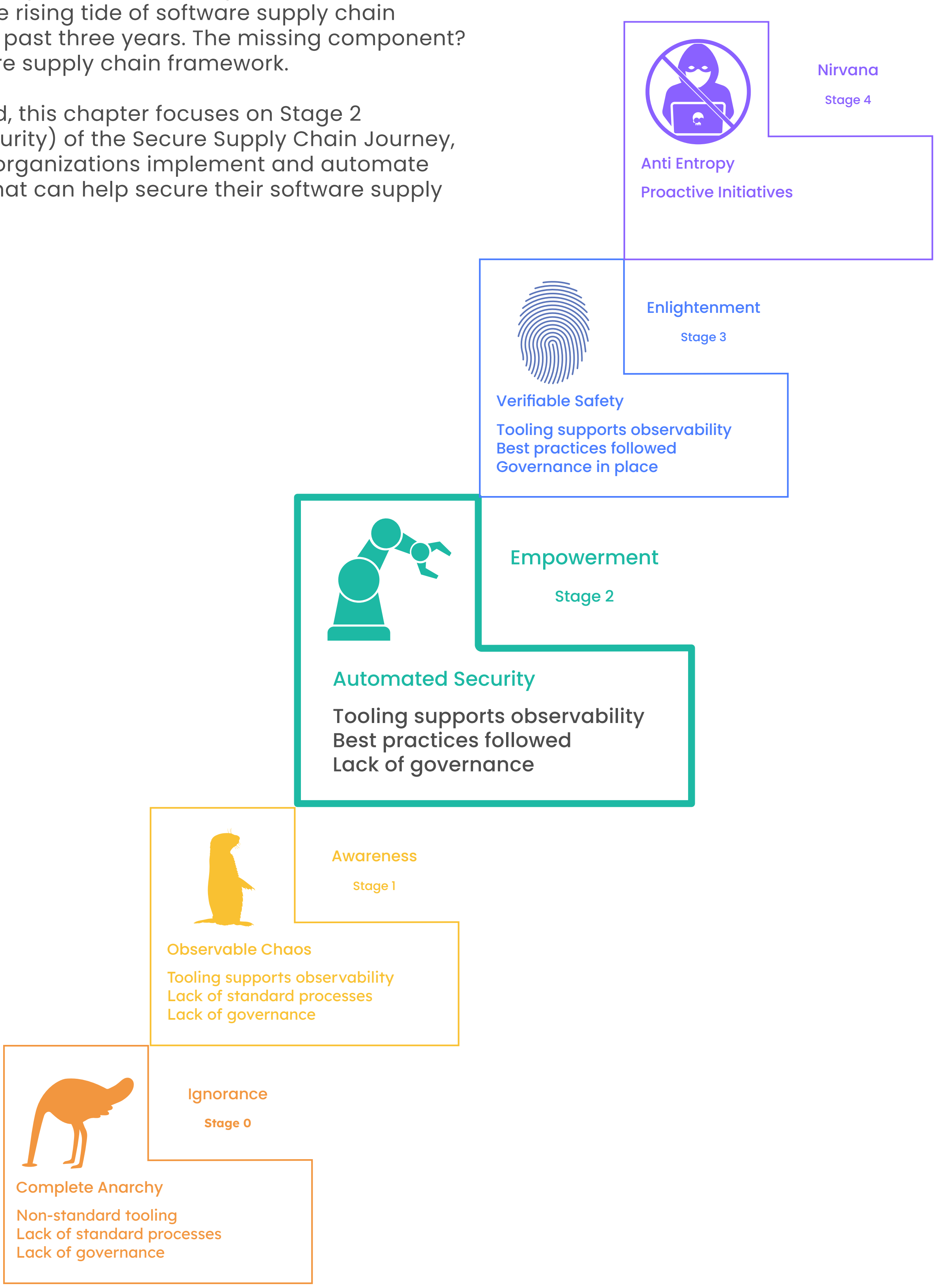
For software developers, the most practical reference document is "Securing the Software Supply Chain: Recommended Practices for Developers", which details a best-practices approach to implementing a Secure Software Development Framework (SSDF), including:

- **Architecture & Design Review –** developers, suppliers and customers must work together to define software requirements up front.
- **Software Threat Modeling –** security architects should develop threat models for all critical components and systems.
- **Coding Standards –** standard coding best practices apply.
- **Secure Library Checks –** incorporate only third-party/open source libraries that have been vetted by your organization.
- **Code & Executable Testing –** use Static & Dynamic Application Security Testing (SAST and DAST) apps, as well as Software Composition Analysis (SCA) tools to identify issues.
- **Secure Build & Delivery –** harden the development, build and delivery environments.

If you're like most mid-sized companies (or larger) that characterize this stage on the Secure Supply Chain Journey, these best practices shouldn't come as a surprise. They've been championed in one form or another for years. They also contain overlapping requirements with existing certifications and standards such as SOC2, PCI-DSS, ISO 27001, etc. which means you're very likely to have a number of these best practices already in place.

Unfortunately, despite the fact that SSDFs have been with us for decades, software vendors both big and small (from Solarwinds to Kaseya to most recently CircleCI) were still blindsided by the rising tide of software supply chain attacks over the past three years. The missing component? A secure software supply chain framework.

With that in mind, this chapter focuses on Stage 2 (Automated Security) of the Secure Supply Chain Journey, which can help organizations implement and automate best practices that can help secure their software supply chain.



“[Software vendors] must be held liable when they fail to live up to the duty of care they owe consumers, businesses, or critical infrastructure providers.”

– US Government Administration proposing new legislation to establish liability for software vendors

Empowerment
Stage 2
Automated Security

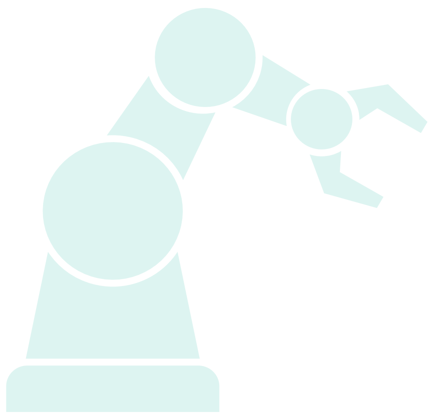
The second step on the journey to a secure software supply chain involves implementing a supply chain framework. While the SCA and SBOM tools discussed in Stage 1 provide transparency into what's in your software, the tools and processes outlined here can help ensure that the third-party code you import into your organization, as well as the code you build and use are secure.

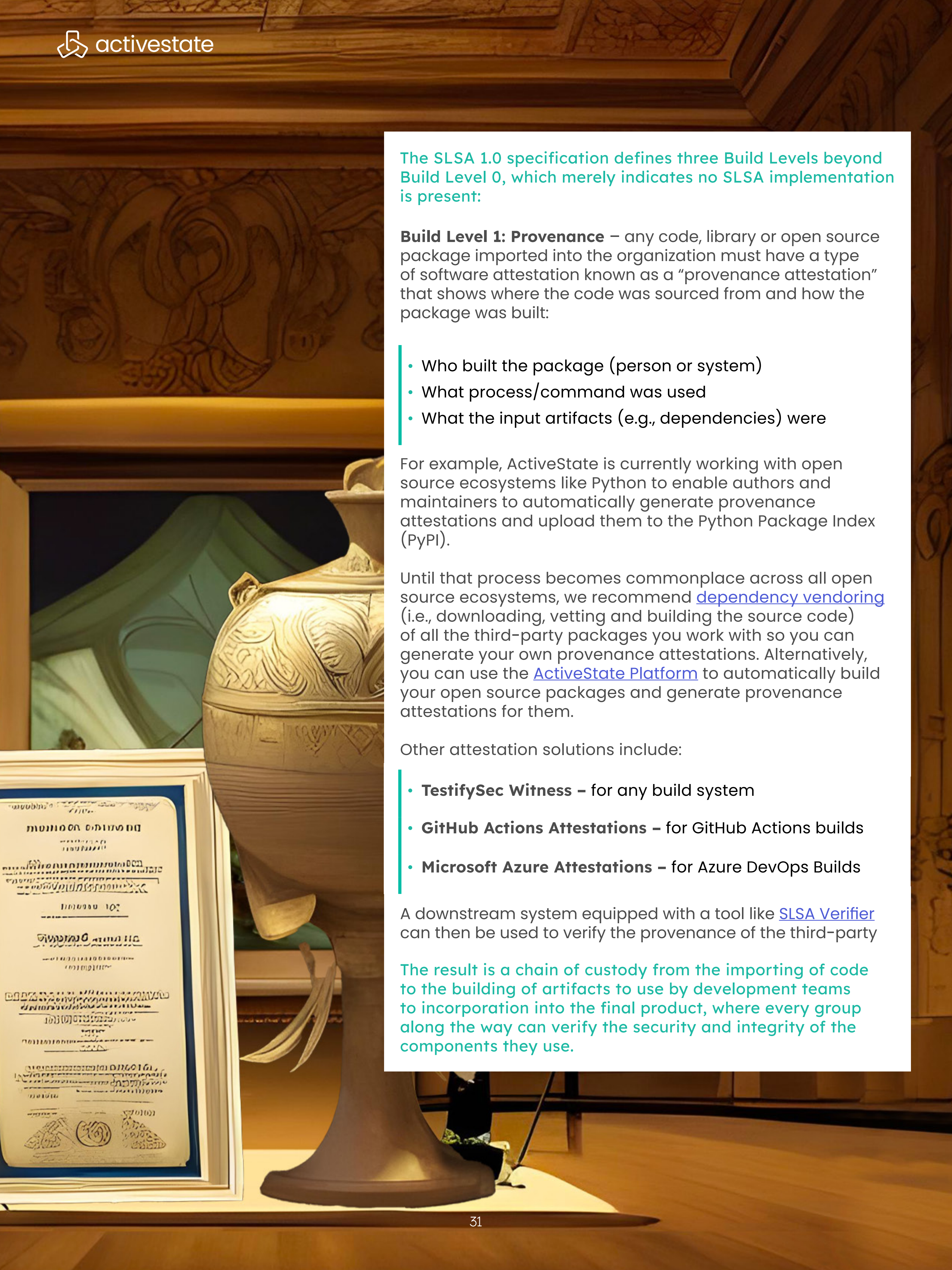
Characteristics include:

- **Standard Tooling -** implement verifiable controls that identify the provenance (i.e., the source) of the third-party packages you import and build.
- **Standard Processes -** implement secure software supply chain best practices when importing third-party code, as well as when building software artifacts.
- **Lack of Governance -** a governance layer to ensure best practices are followed is not required at this stage.

While we certainly don't advocate abandoning any SSDF standards you may have already implemented, you will need to incorporate a software supply chain security framework if you hope to avoid the growing tsunami of supply chain attacks, as well as ensure you can comply with US government supplier requirements.

The Supply chain Levels for Software Artifacts (SLSA) security framework, long used by Google, has recently been brought to market by the Open Source Software Foundation (OSSF) in conjunction with a consortium of industry collaborators. Implementing its controls and best practices can help ensure that the code you import and/or build in your organization is done in a secure manner.





The SLSA 1.0 specification defines three Build Levels beyond Build Level 0, which merely indicates no SLSA implementation is present:

Build Level 1: Provenance – any code, library or open source package imported into the organization must have a type of software attestation known as a "provenance attestation" that shows where the code was sourced from and how the package was built:

- Who built the package (person or system)
- What process/command was used
- What the input artifacts (e.g., dependencies) were

For example, ActiveState is currently working with open source ecosystems like Python to enable authors and maintainers to automatically generate provenance attestations and upload them to the Python Package Index (PyPI).

Until that process becomes commonplace across all open source ecosystems, we recommend dependency vendoring (i.e., downloading, vetting and building the source code) of all the third-party packages you work with so you can generate your own provenance attestations. Alternatively, you can use the ActiveState Platform to automatically build your open source packages and generate provenance attestations for them.

Other attestation solutions include:

- **TestifySec Witness** – for any build system
- **GitHub Actions Attestations** – for GitHub Actions builds
- **Microsoft Azure Attestations** – for Azure DevOps Builds

A downstream system equipped with a tool like SLSA Verifier can then be used to verify the provenance of the third-party

The result is a chain of custody from the importing of code to the building of artifacts to use by development teams to incorporation into the final product, where every group along the way can verify the security and integrity of the components they use.

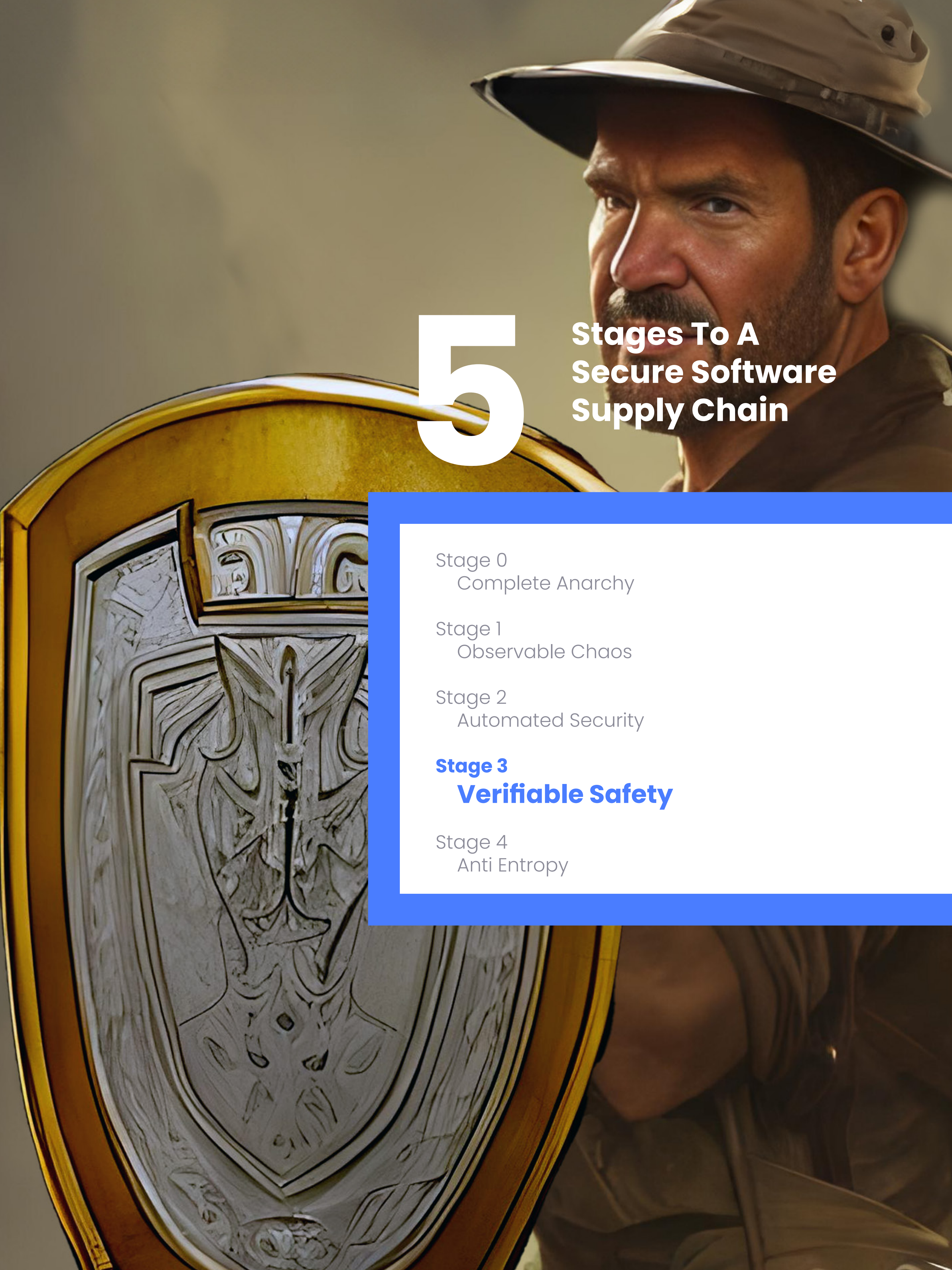
Build Level 2: Build Service – the only way to ensure the security of the software you work with and produce is to build it yourself from source code. Accordingly, Level 2 introduces not only a build service but also a signing service to ensure that neither the package nor the provenance attestation generated for it were tampered with after being created. A downstream service would then verify the authenticity of the signatures.

Build Level 3: Hardened Builds – to help ensure the build service cannot be compromised (such as happened with Solarwinds), a number of controls should be put in place to harden the build service, including:

- Implement pre-scripted, parameterless builds to ensure hackers can't get access to/edit build scripts.
- Create build environments that are ephemeral, isolated, and hermetically sealed (i.e., no access to the internet) to ensure against corrupted environments and/or hackers compromising the build process.
- Isolate the signing service introduced in Build Level 2 to ensure hackers can't access secrets used to sign the provenance.

Implementing SSDF and SLSA best practices will go a long way to securing your software development process from end to end, but even automated processes can be bypassed.

The next stage in the journey to software supply chain security focuses on implementing a governance layer to ensure that the best practices you've put in place are actually followed.



5 Stages To A Secure Software Supply Chain

Stage 0
Complete Anarchy

Stage 1
Observable Chaos

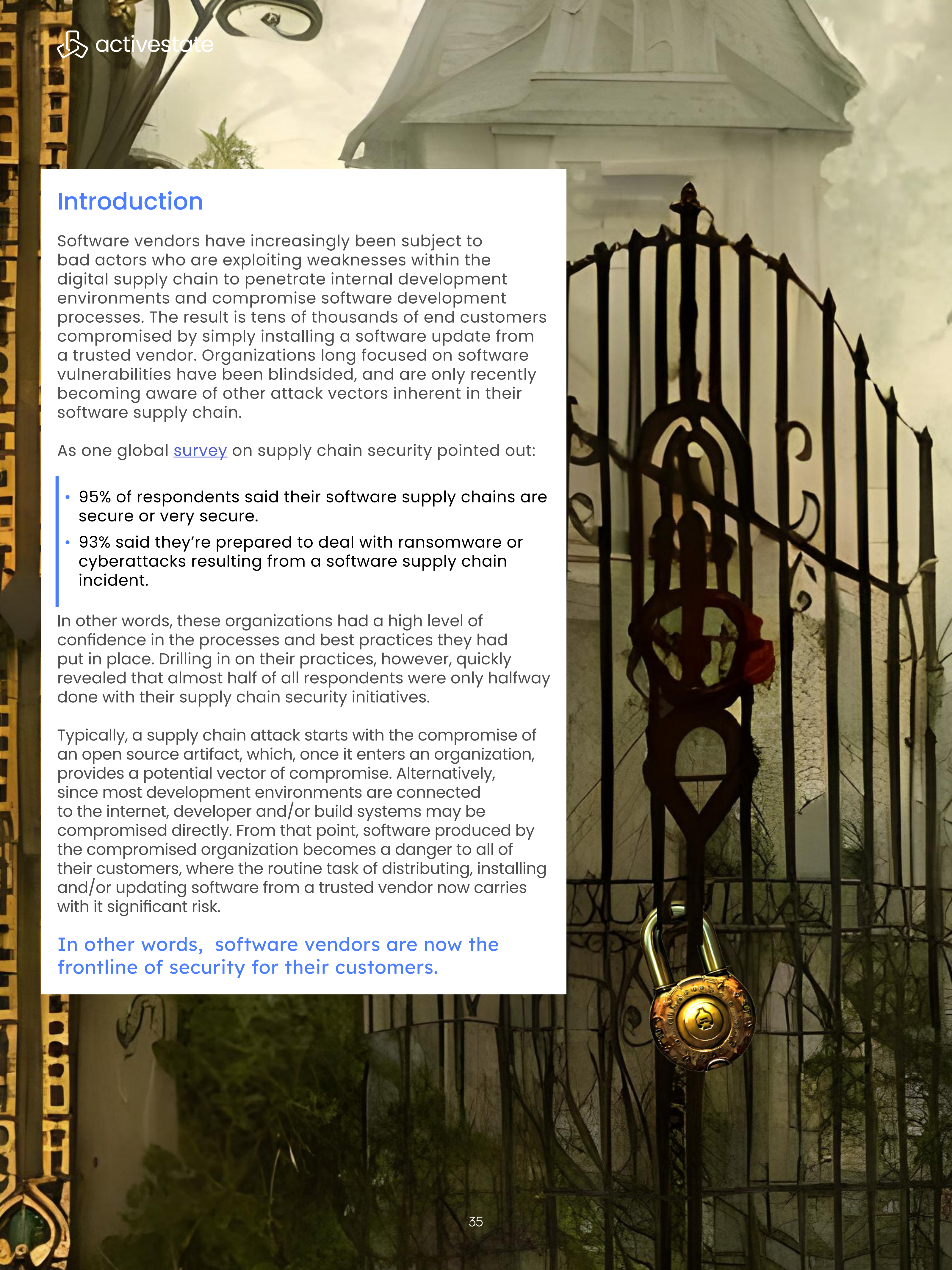
Stage 2
Automated Security

Stage 3
Verifiable Safety

Stage 4
Anti Entropy

“*Trust but verify*”

-Ronald Reagan



Introduction

Software vendors have increasingly been subject to bad actors who are exploiting weaknesses within the digital supply chain to penetrate internal development environments and compromise software development processes. The result is tens of thousands of end customers compromised by simply installing a software update from a trusted vendor. Organizations long focused on software vulnerabilities have been blindsided, and are only recently becoming aware of other attack vectors inherent in their software supply chain.

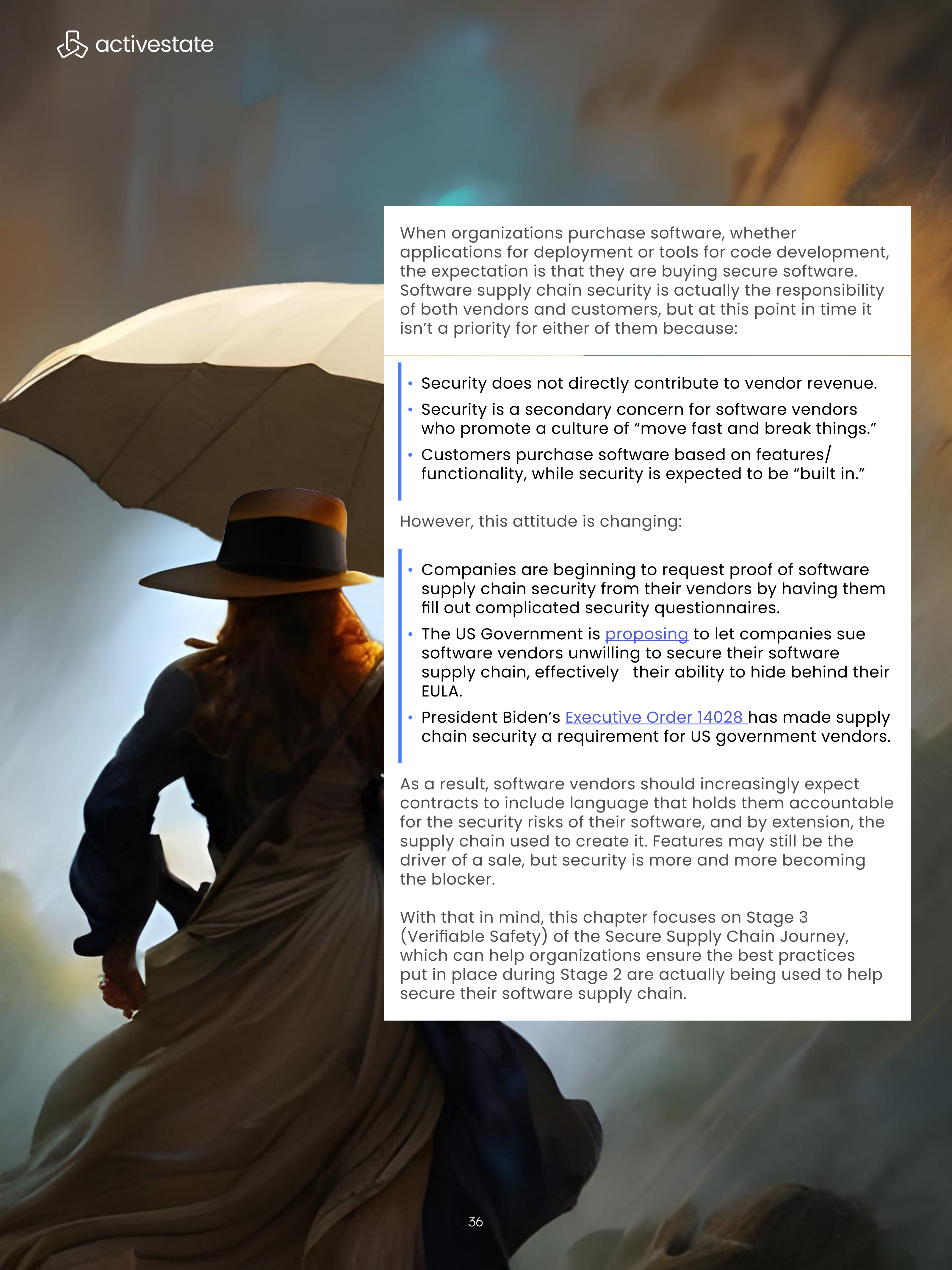
As one global survey on supply chain security pointed out:

- 95% of respondents said their software supply chains are secure or very secure.
- 93% said they're prepared to deal with ransomware or cyberattacks resulting from a software supply chain incident.

In other words, these organizations had a high level of confidence in the processes and best practices they had put in place. Drilling in on their practices, however, quickly revealed that almost half of all respondents were only halfway done with their supply chain security initiatives.

Typically, a supply chain attack starts with the compromise of an open source artifact, which, once it enters an organization, provides a potential vector of compromise. Alternatively, since most development environments are connected to the internet, developer and/or build systems may be compromised directly. From that point, software produced by the compromised organization becomes a danger to all of their customers, where the routine task of distributing, installing and/or updating software from a trusted vendor now carries with it significant risk.

In other words, software vendors are now the frontline of security for their customers.



When organizations purchase software, whether applications for deployment or tools for code development, the expectation is that they are buying secure software. Software supply chain security is actually the responsibility of both vendors and customers, but at this point in time it isn't a priority for either of them because:

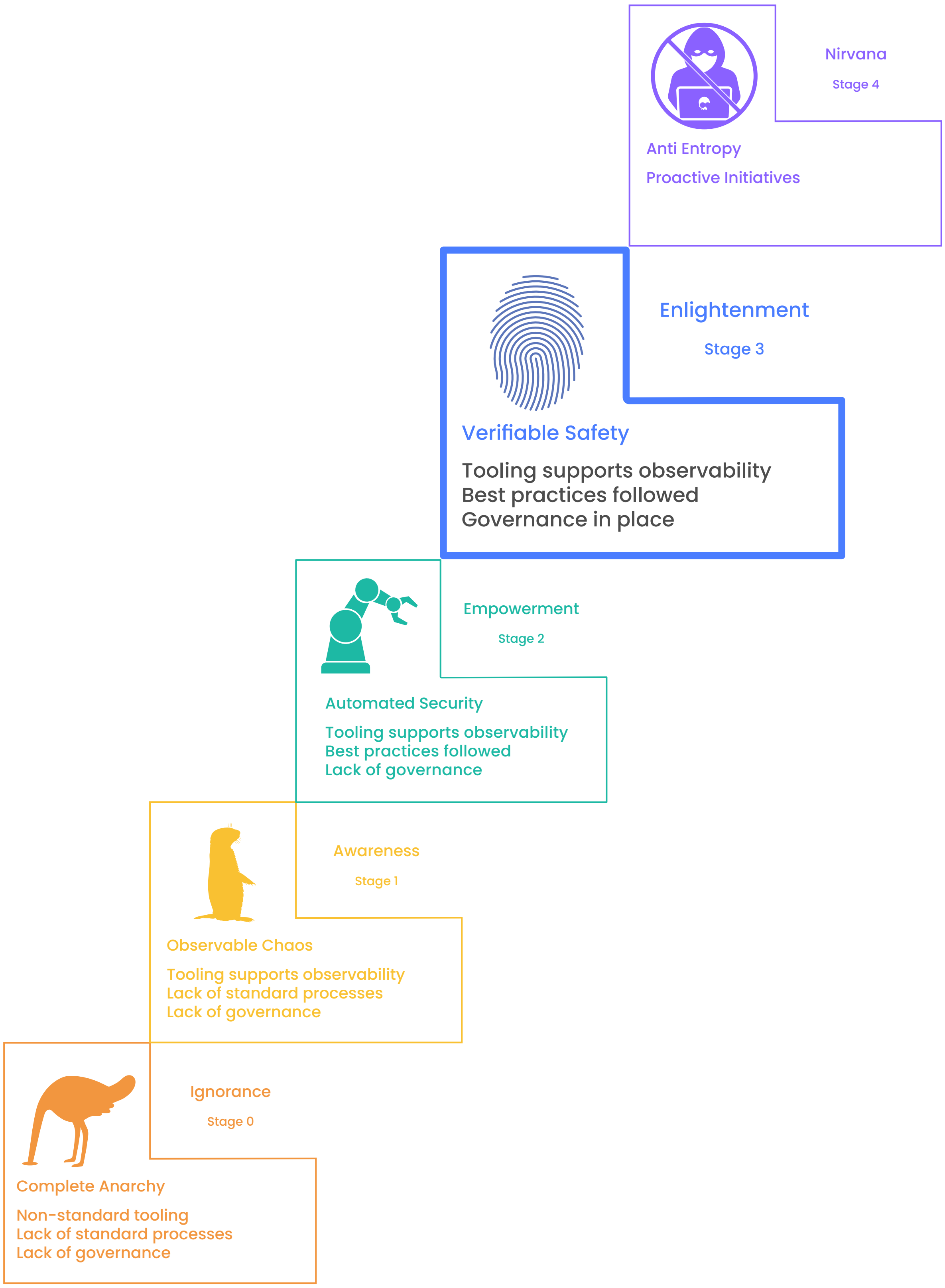
- Security does not directly contribute to vendor revenue.
- Security is a secondary concern for software vendors who promote a culture of "move fast and break things."
- Customers purchase software based on features/ functionality, while security is expected to be "built in."

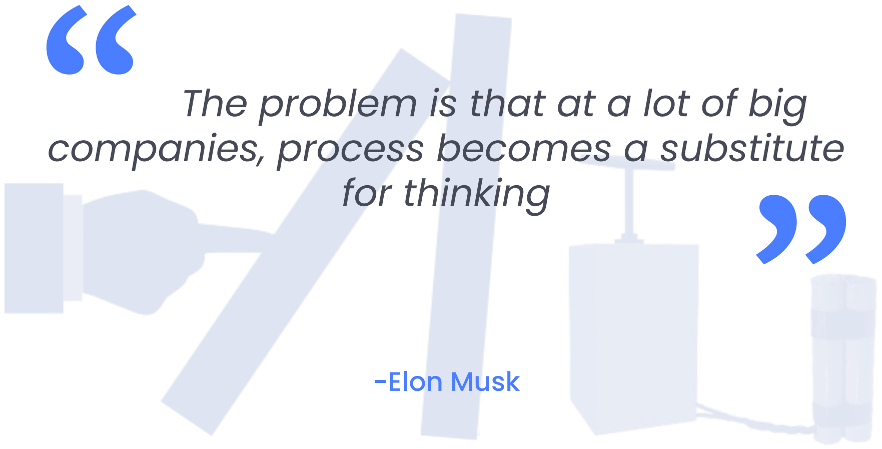
However, this attitude is changing:

- Companies are beginning to request proof of software supply chain security from their vendors by having them fill out complicated security questionnaires.
- The US Government is proposing to let companies sue software vendors unwilling to secure their software supply chain, effectively   their ability to hide behind their EULA.
- President Biden's Executive Order 14028 has made supply chain security a requirement for US government vendors.

As a result, software vendors should increasingly expect contracts to include language that holds them accountable for the security risks of their software, and by extension, the supply chain used to create it. Features may still be the driver of a sale, but security is more and more becoming the blocker.

With that in mind, this chapter focuses on Stage 3 (Verifiable Safety) of the Secure Supply Chain Journey, which can help organizations ensure the best practices put in place during Stage 2 are actually being used to help secure their software supply chain.


Nirvana
Stage 4
Anti Entropy
Proactive Initiatives
Enlightenment
Stage 3
Verifiable Safety
Tooling supports observability
Best practices followed
Governance in place
Empowerment
Stage 2
Automated Security
Tooling supports observability
Best practices followed
Lack of governance
Awareness
Stage 1
Observable Chaos
Tooling supports observability
Lack of standard processes
Lack of governance
Ignorance
Stage 0
Complete Anarchy
Non-standard tooling
Lack of standard processes
Lack of governance



"The problem is that at a lot of big companies, process becomes a substitute for thinking"

-Elon Musk

Enlightenment
Stage 3
Verifiable Safety

The third stage on the journey to a secure software supply chain involves implementing governance controls to ensure that best practices discussed in Stage 2 are being followed. Without enforcement, best practices all too often amount to nothing more than good intentions leaving your processes exposed to the elements that can erode them. The tools and processes in this stage can help ensure that your best practices are actually practiced.

Characteristics include:

- **Standard Tooling -** implement enforcement tooling, such as a policy engine which can enforce predefined rules.
- **Standard Practices -** implement secure software supply chain best practices when importing third-party code, as well as when building software artifacts.
- **Governance -** implement a governance layer to enforce best practices and ensure supply chain security criteria are met.

The simplest way to meet your goals is to deploy a flexible policy engine as part of your automated processes. By defining rules in the policy engine and placing it at key points in your software development processes, you can enforce how and when software supply chain security goals are achieved.

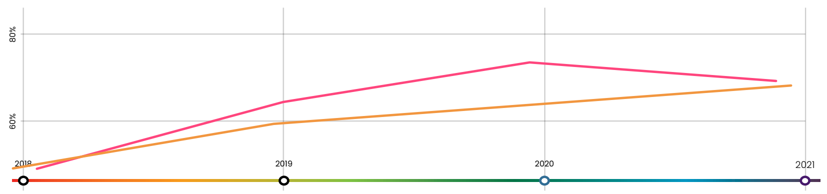
Depending on your specific software development processes, there are likely a number of areas that could benefit from the strong governance a policy engine can offer. Here, We'll focus on three problem areas that most enterprises wrestle with when it comes to securing their software supply chain.

Codebase Integrity

Every enterprise has best practices around updating and remediating their codebase. Unfortunately, as surveys continue to show, these processes are rarely followed:

78 Of code in codebases was open source

81% Contained at least one vulnerability



88% Contained components that had no new development in two years

85% Contained open source that was more than four years out-of-date

Source: Open Source Security and Risk Analysis Report

And as the State of Software Security report reminds us:

"Open source libraries are constantly evolving: what appears secure today may not be tomorrow. Despite this dynamic landscape **70 percent** of the time, developers never update third-party libraries after including them in a codebase."

A codebase that is rarely updated will contain more vulnerabilities, bugs and performance issues over time, posing a greater risk to anyone that runs it. But the tradeoff is development time and resources, which are assumed to be better spent on new features/functionality. Besides, nobody wants to be accused of breaking the build should an upgrade effort not go smoothly.

Instead of a "big bang" approach, consider devoting 10-20% of each development sprint to addressing outdated and vulnerable packages. One of the best areas to enforce this behavior is in your artifact repository, which may already have governance capabilities built in. If not, you can use a policy engine to flag:

- **Package datedness -** ie., disallow use of packages greater than X months old
- **Package vulnerability -** i.e., disallow use of packages with a severity rating greater than or equal to Y

A second process where governance can be used to enforce codebase integrity is during CI/CD environment creation, which brings us back to SBOMs and Attestations.

Container Integrity

While Software Bills of Material (SBOMs) are relatively new, many enterprises already have the capability to generate them. Unfortunately, once generated, they rarely do anything with them, which is a shame since they can act as a key enforcement mechanism.

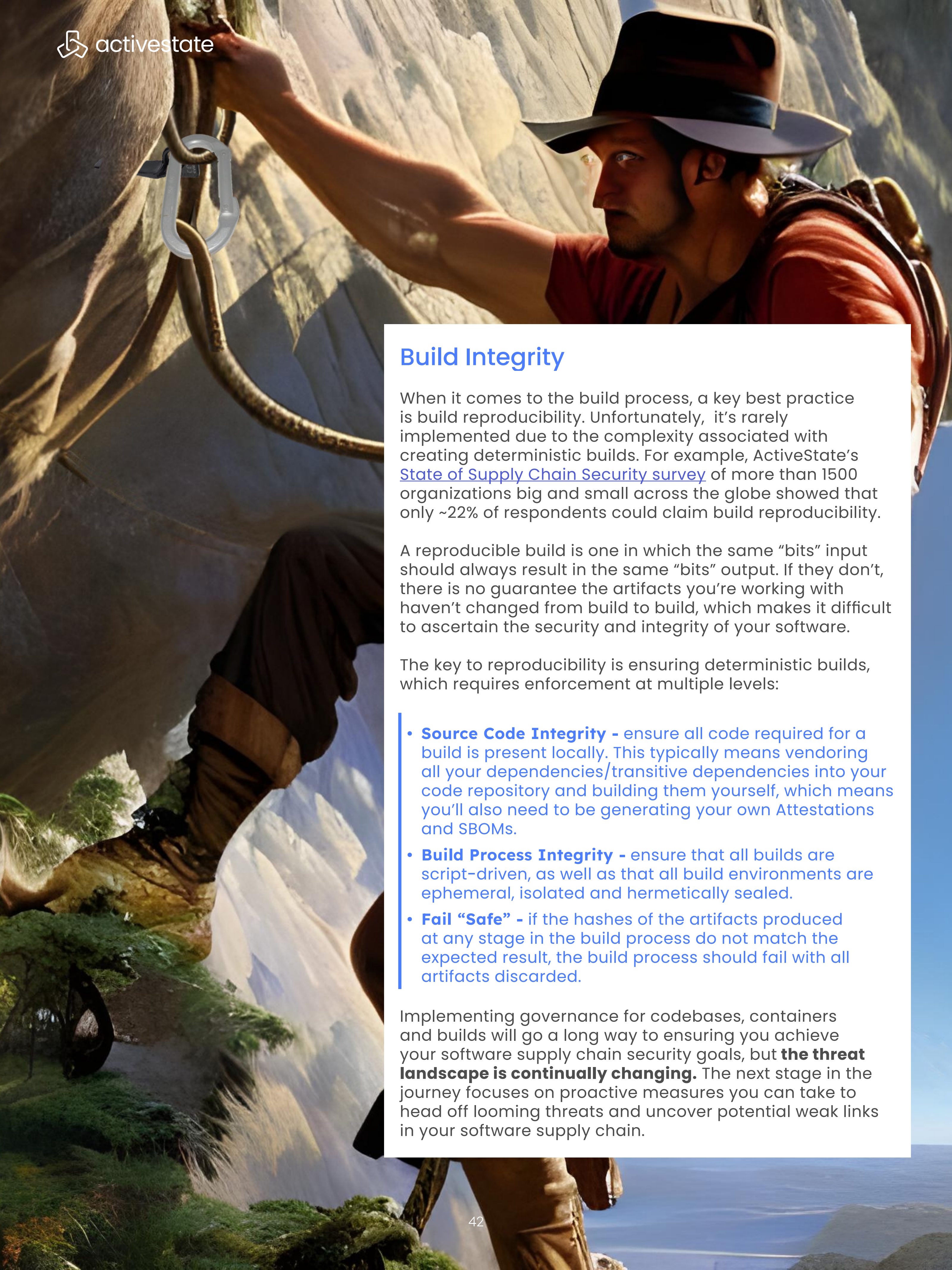
For example, containers used in the CI/CD process often get out of date, but SBOMs can ensure that:

- Container runtime environments are built with correct versions of the correct open source packages, as well as ensure all required packages are present - no more, and no less.
- No packages feature vulnerabilities of a severity level that the organization considers a threat.
  - To ensure against false positives, SBOM specifications like [CycloneDX]() and [SPDX]() include metadata (such as Vulnerability Exploitability eXchange or VEX data) that lets developers specify whether shipped vulnerabilities are actually exploitable.

Similarly:

- Provenance Attestations can be checked to ensure code has been sourced correctly.
- Verification Summary Attestations (VSAs) can be checked to ensure whether prebuilt packages/artifacts have been built in a secure manner.

While container integrity is key, the software produced by the CI/CD process is only as secure as the weakest link in the build process, which means we need to talk about build reproducibility.



Build Integrity

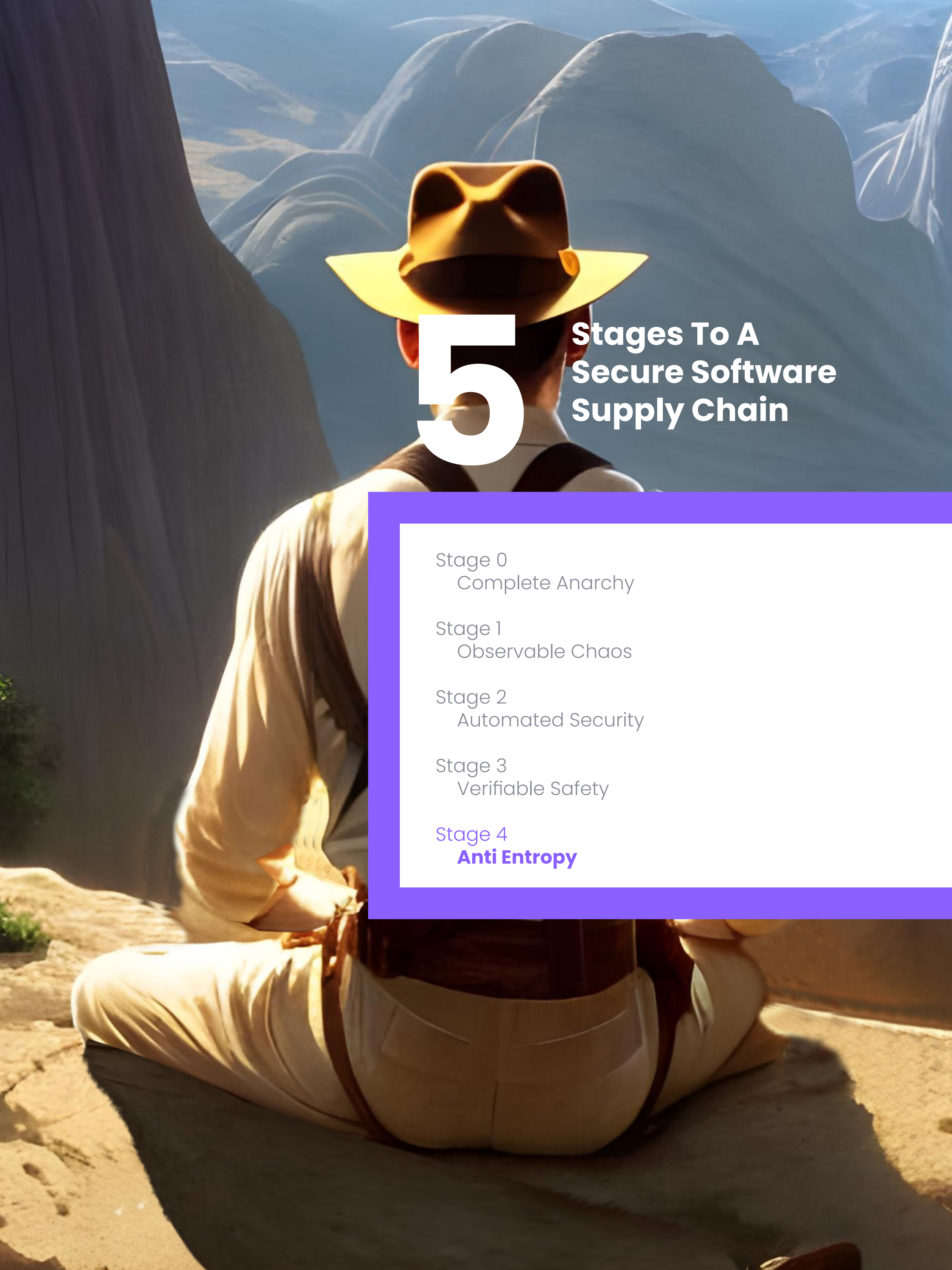
When it comes to the build process, a key best practice is build reproducibility. Unfortunately, it's rarely implemented due to the complexity associated with creating deterministic builds. For example, ActiveState's State of Supply Chain Security survey of more than 1500 organizations big and small across the globe showed that only ~22% of respondents could claim build reproducibility.

A reproducible build is one in which the same "bits" input should always result in the same "bits" output. If they don't, there is no guarantee the artifacts you're working with haven't changed from build to build, which makes it difficult to ascertain the security and integrity of your software.

The key to reproducibility is ensuring deterministic builds, which requires enforcement at multiple levels:

- **Source Code Integrity -** ensure all code required for a build is present locally. This typically means vendoring all your dependencies/transitive dependencies into your code repository and building them yourself, which means you'll also need to be generating your own Attestations and SBOMs.
- **Build Process Integrity -** ensure that all builds are script-driven, as well as that all build environments are ephemeral, isolated and hermetically sealed.
- **Fail "Safe" -** if the hashes of the artifacts produced at any stage in the build process do not match the expected result, the build process should fail with all artifacts discarded.

Implementing governance for codebases, containers and builds will go a long way to ensuring you achieve your software supply chain security goals, but **the threat landscape is continually changing.** The next stage in the journey focuses on proactive measures you can take to head off looming threats and uncover potential weak links in your software supply chain.



5 Stages To A Secure Software Supply Chain

Stage 0
Complete Anarchy

Stage 1
Observable Chaos

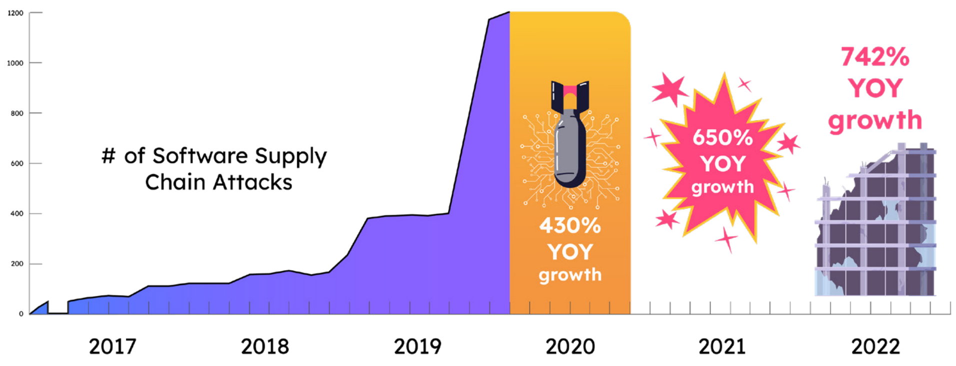
Stage 2
Automated Security

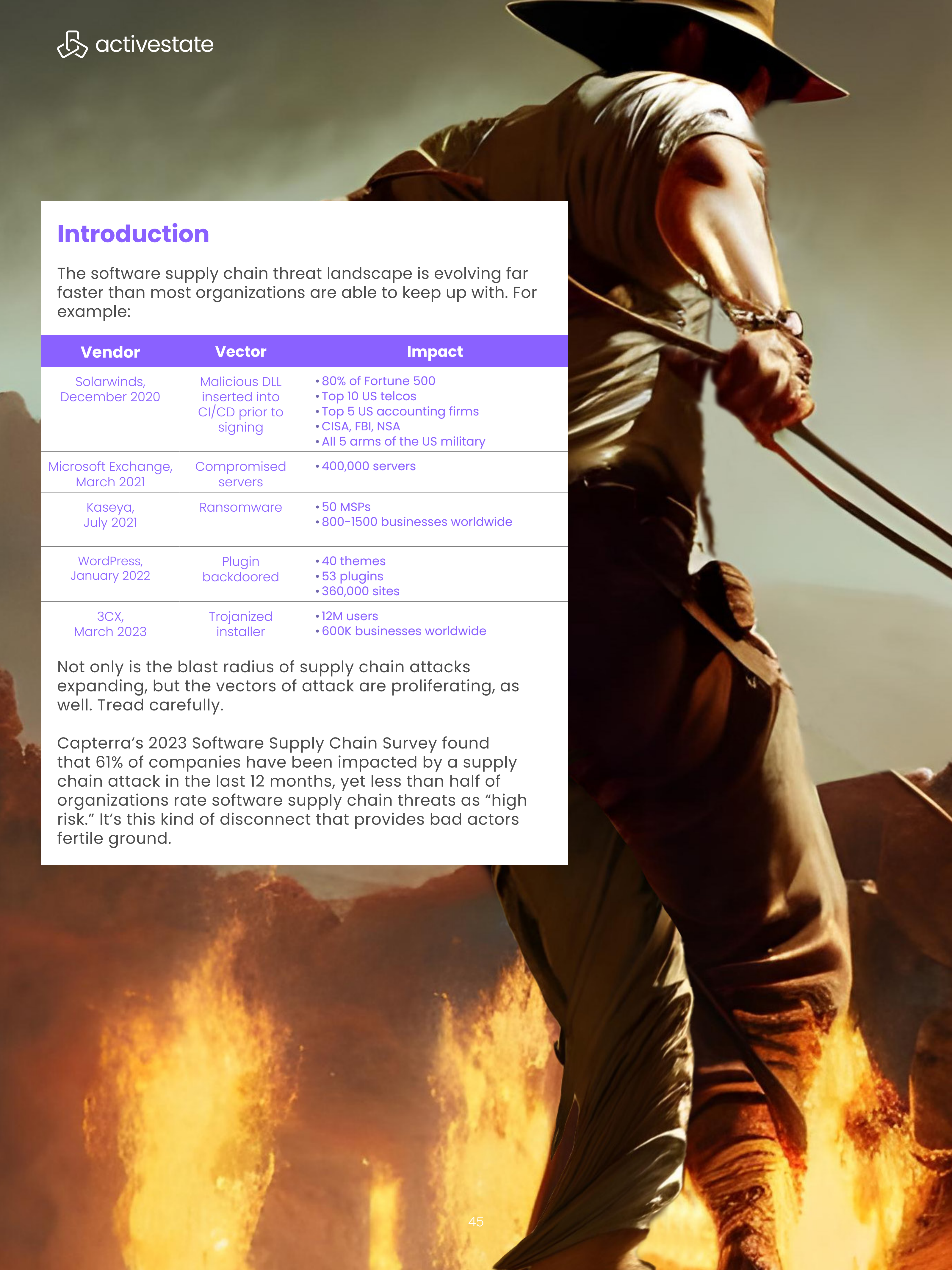
Stage 3
Verifiable Safety

Stage 4
Anti Entropy

“ *The private sector must adapt to the continuously changing threat environment, [and] ensure its products are built and operate securely.* ”

– President Biden’s Executive Order 14028





Introduction

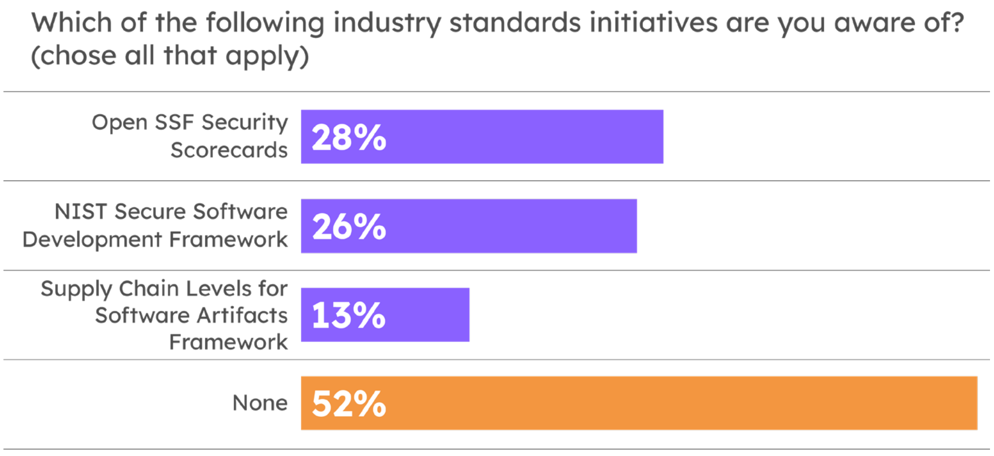
The software supply chain threat landscape is evolving far faster than most organizations are able to keep up with. For example:

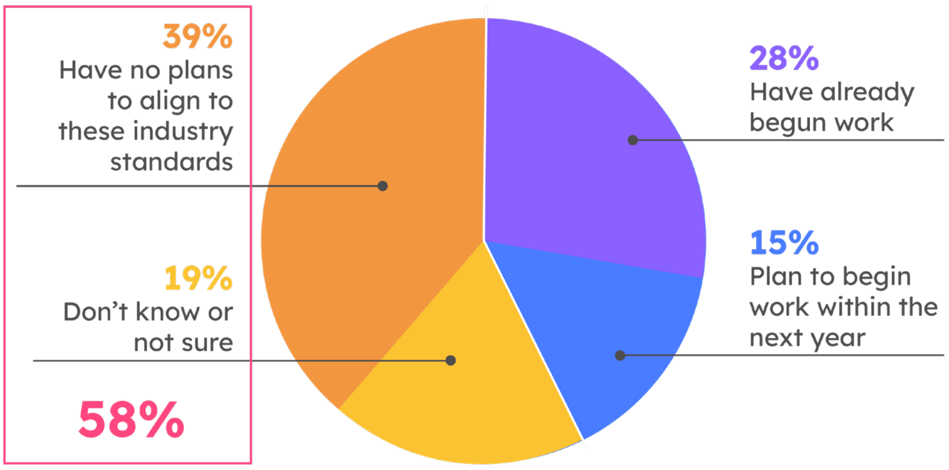
Vendor	Vector	Impact
Solarwinds, December 2020	Malicious DLL inserted into CI/CD prior to signing	• 80% of Fortune 500 • Top 10 US telcos • Top 5 US accounting firms • CISA, FBI, NSA • All 5 arms of the US military
Microsoft Exchange, March 2021	Compromised servers	• 400,000 servers
Kaseya, July 2021	Ransomware	• 50 MSPs • 800-1500 businesses worldwide
WordPress, January 2022	Plugin backdoored	• 40 themes • 53 plugins • 360,000 sites
3CX, March 2023	Trojanized installer	• 12M users • 600K businesses worldwide

Not only is the blast radius of supply chain attacks expanding, but the vectors of attack are proliferating, as well. Tread carefully.

Capterra's 2023 Software Supply Chain Survey found that 61% of companies have been impacted by a supply chain attack in the last 12 months, yet less than half of organizations rate software supply chain threats as "high risk." It's this kind of disconnect that provides bad actors fertile ground.

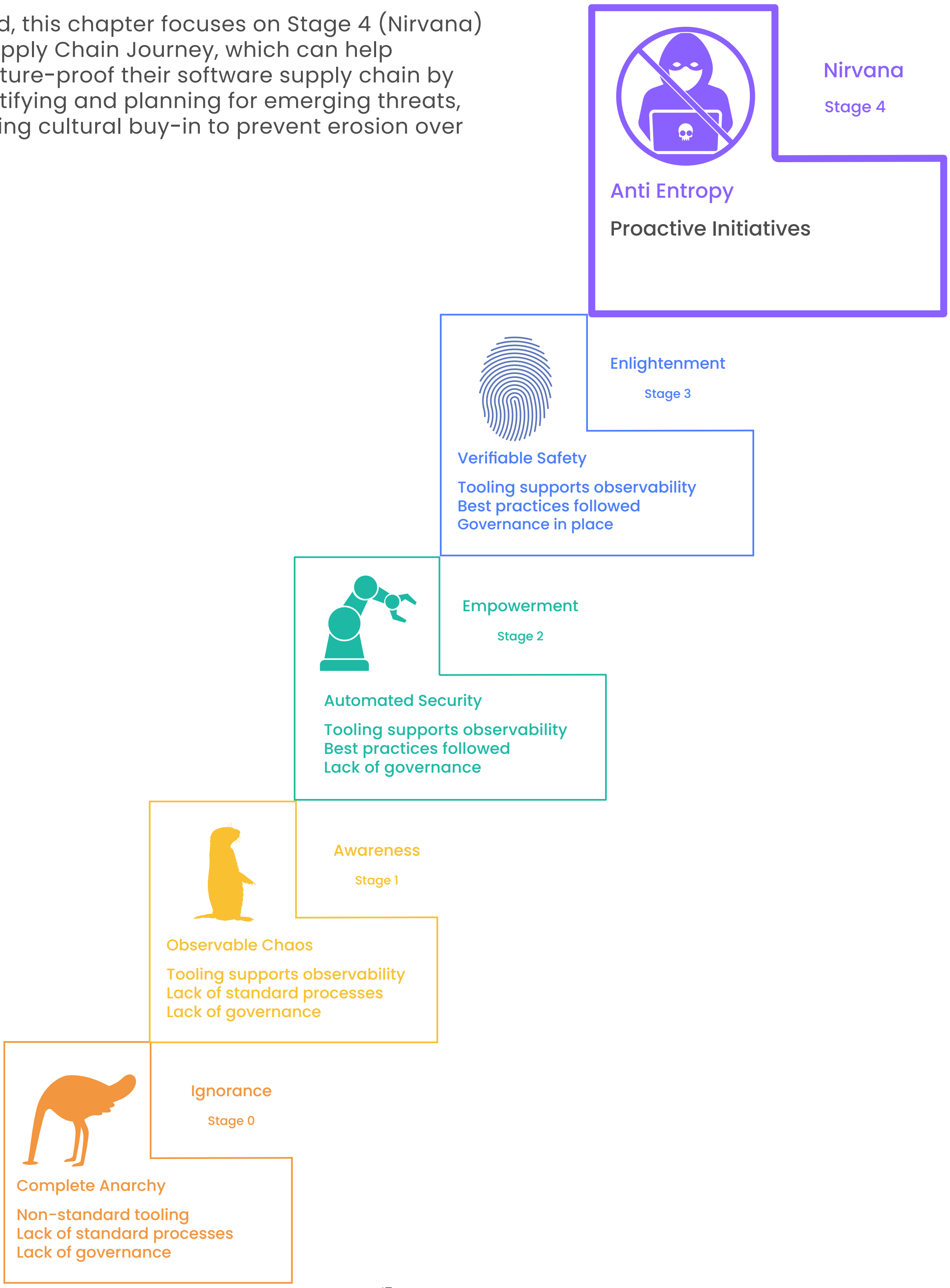
Unfortunately, it's unreasonable to expect the greater community of overworked and unpaid open source authors to close the holes. According to a recent Tidelift survey, maintainers of open source software have enough on their plate without having to worry about supply chain security:





All of which means that even if you have a plan and are well on your way to implementing it, you'll need to start thinking about how to avoid being blindsided by the quickly evolving threats across the supply chain landscape. After all, while best practices evolve over time, so does hacker ingenuity.

With that in mind, this chapter focuses on Stage 4 (Nirvana) of the Secure Supply Chain Journey, which can help organizations future-proof their software supply chain by proactively identifying and planning for emerging threats, as well as ensuring cultural buy-in to prevent erosion over time.



“*74% of IT pros believe technologies like static and dynamic application security testing [SAST & DAST] are important, but feel that those technologies aren’t enough to protect them from supply chain threats*”

Reversinglabs

Nirvana
Stage 4
Anti EntroZpy

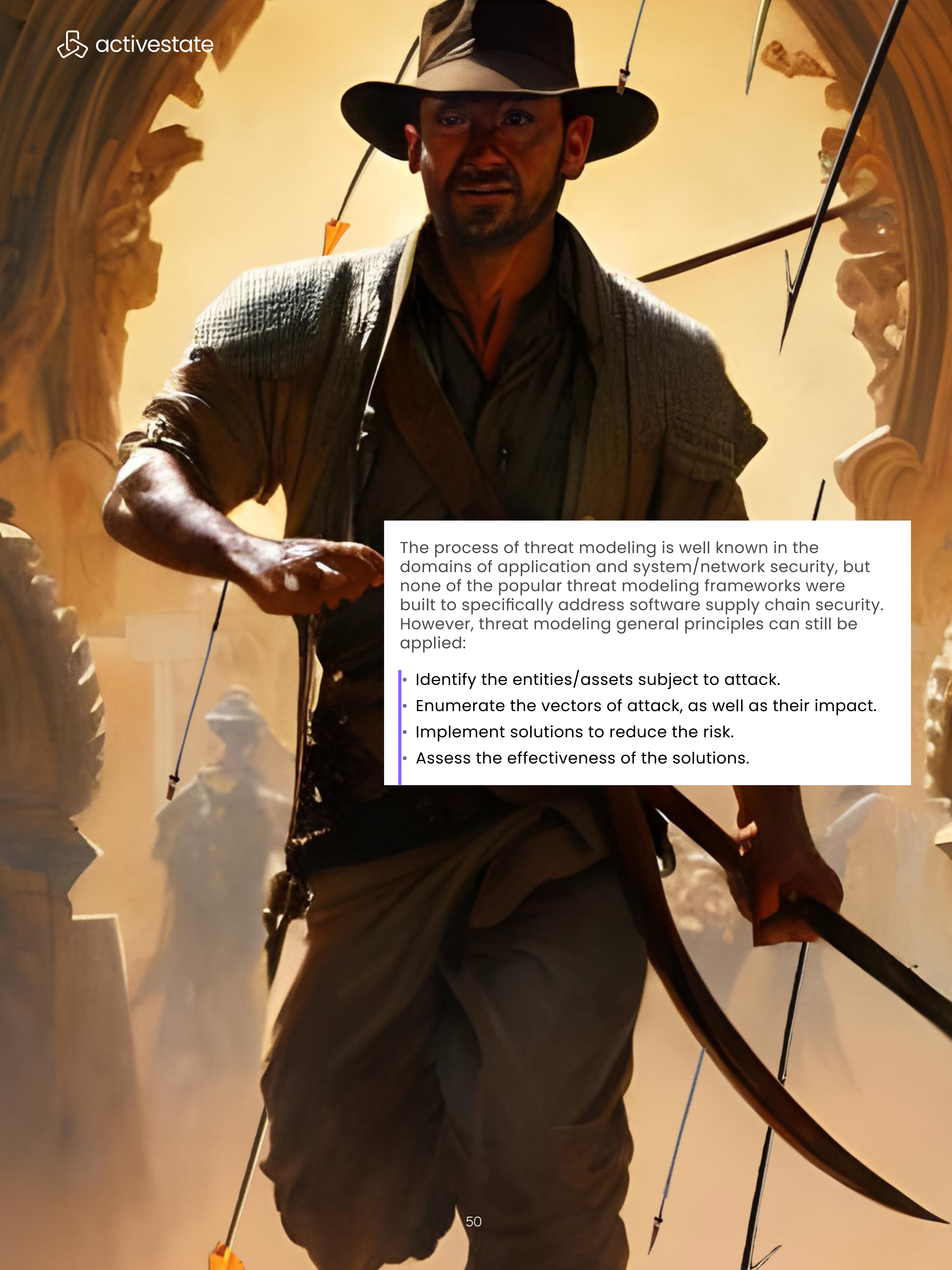
If you've reached the fourth stage on the journey to a secure software supply chain, take a moment to celebrate the accomplishment. Not only do you now know where all the skeletons in your supply chain live, but you've got the best practices in place to deal with them, and the governance to ensure they don't accidentally come back to haunt you. No mean feat at a time when the cost of software supply chain attacks is expected to exceed $45B.

Having reached the pinnacle of your journey all that's left to do is make sure you can't easily be toppled off. That means getting a handle on existing and emerging threats, as well as ensuring your controls are resilient enough to withstand them. But it also means fostering a culture that internalizes those needs, as well.

Some of the tools and practices that can help with Stage 4 include:

- **Standard Tooling -** implement threat modeling tooling that can help visualize systems, flows and vectors of attack.
- **Standard Practices -** simulate and analyze the effect of attacks, both when key controls are present and when they're not in order to assess the effectiveness of/need for redundancy.
- **Governance** - implement a culture of software supply chain security that truly makes it everyone's responsibility.

Keep in mind that your software supply chain is only as strong as its weakest link, which is constantly being redefined as new vulnerabilities are discovered and hackers explore new tools, targets and tactics. To keep up, you'll need a repeatable process that can help identify threats and evaluate the effectiveness of existing systems/controls.

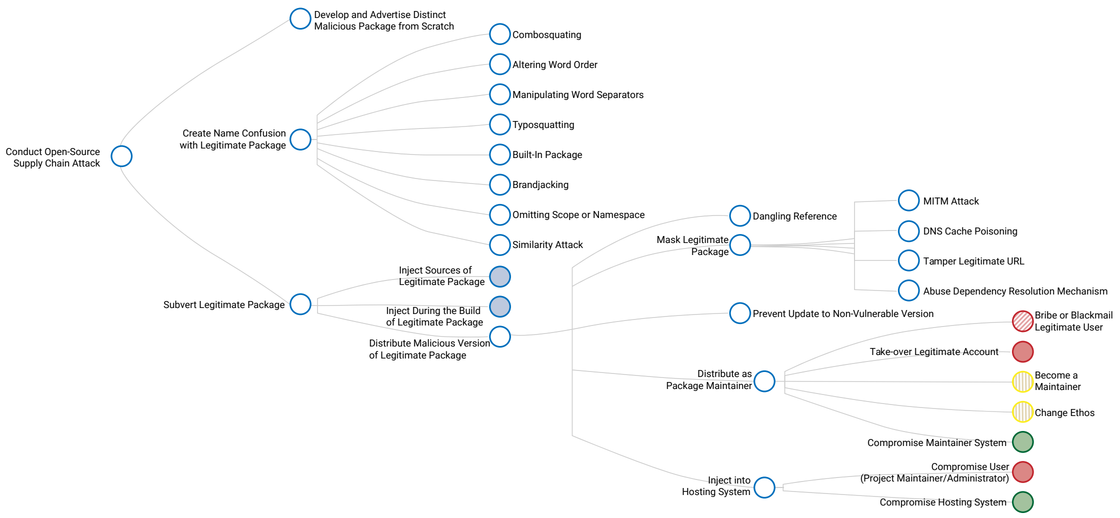


The process of threat modeling is well known in the domains of application and system/network security, but none of the popular threat modeling frameworks were built to specifically address software supply chain security. However, threat modeling general principles can still be applied:

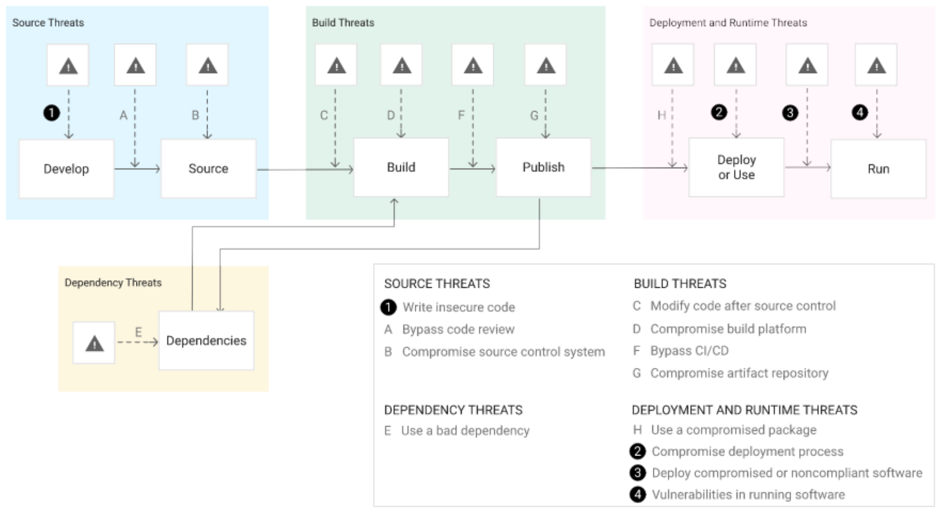
- Identify the entities/assets subject to attack.
- Enumerate the vectors of attack, as well as their impact.
- Implement solutions to reduce the risk.
- Assess the effectiveness of the solutions.

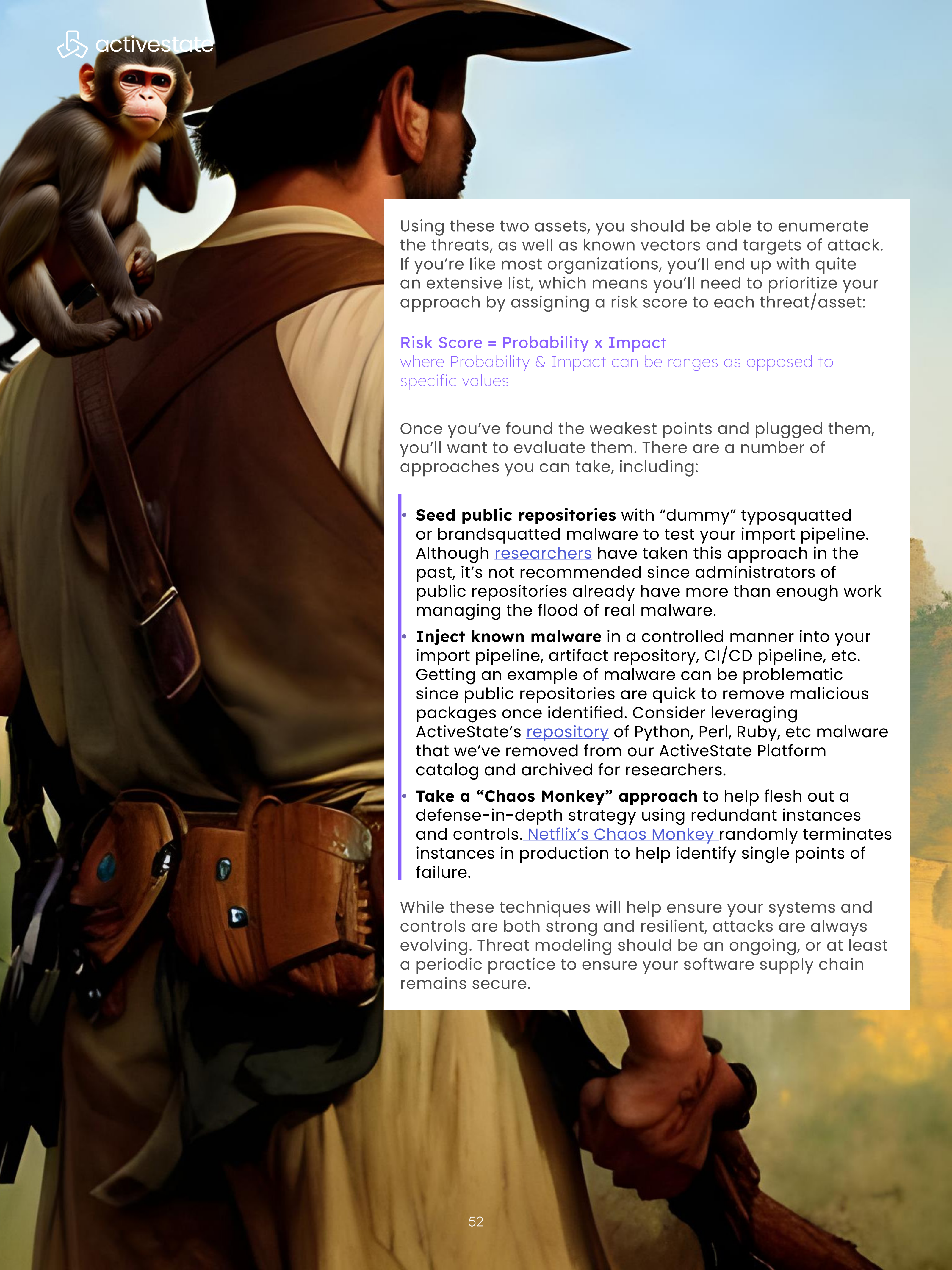
Because the software supply chain is both wide and deep, it may be easier to identify vulnerable entities by enumerating known attacks and their targets. There are two useful approaches here:

- SAP's software supply chain risk explorer provides an interactive attack tree, starting with abstract, top-level goals and drilling down to identify known attack methods and techniques. For example:


Conduct Open-Source Supply Chain Attack
Develop and Advertise Distinct Malicious Package from Scratch
Create Name Confusion with Legitimate Package
Combosquatting
Altering Word Order
Manipulating Word Separators
Typosquatting
Built-In Package
Brandjacking
Omitting Scope or Namespace
Similarity Attack
Subvert Legitimate Package
Inject Sources of Legitimate Package
Inject During the Build of Legitimate Package
Distribute Malicious Version of Legitimate Package
Mask Legitimate Package
Dangling Reference
MITM Attack
DNS Cache Poisoning
Tamper Legitimate URL
Abuse Dependency Resolution Mechanism
Prevent Update to Non-Vulnerable Version
Distribute as Package Maintainer
Bribe or Blackmail Legitimate User
Take-over Legitimate Account
Become a Maintainer
Change Ethos
Compromise Maintainer System
Inject into Hosting System
Compromise User (Project Maintainer/Administrator)
Compromise Hosting System

- Google's software supply chain threats diagram highlights potentially vulnerable entry points across a typical software development lifecycle:


Source Threats
Build Threats
Deployment and Runtime Threats
1
A
B
C
D
F
G
H
2
3
4
Develop
Source
Build
Publish
Deploy or Use
Run
Dependency Threats
E
Dependencies
SOURCE THREATS
1 Write insecure code
A Bypass code review
B Compromise source control system
BUILD THREATS
C Modify code after source control
D Compromise build platform
F Bypass CI/CD
G Compromise artifact repository
DEPENDENCY THREATS
E Use a bad dependency
DEPLOYMENT AND RUNTIME THREATS
H Use a compromised package
2 Compromise deployment process
3 Deploy compromised or noncompliant software
4 Vulnerabilities in running software



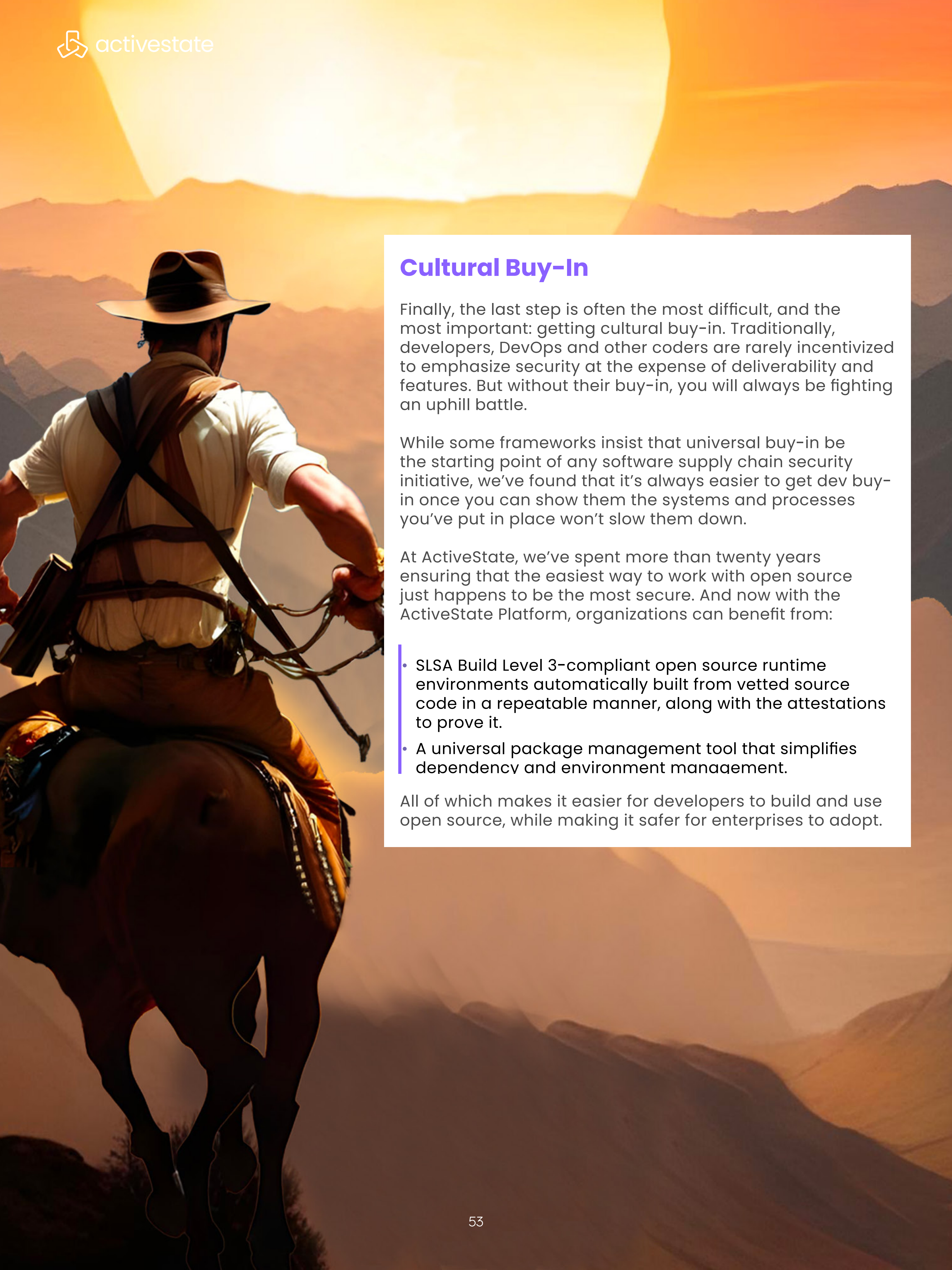
Using these two assets, you should be able to enumerate the threats, as well as known vectors and targets of attack. If you're like most organizations, you'll end up with quite an extensive list, which means you'll need to prioritize your approach by assigning a risk score to each threat/asset:

Risk Score = Probability x Impact
where Probability & Impact can be ranges as opposed to specific values

Once you've found the weakest points and plugged them, you'll want to evaluate them. There are a number of approaches you can take, including:

- **Seed public repositories** with "dummy" typosquatted or brandsquatted malware to test your import pipeline. Although researchers have taken this approach in the past, it's not recommended since administrators of public repositories already have more than enough work managing the flood of real malware.
- **Inject known malware** in a controlled manner into your import pipeline, artifact repository, CI/CD pipeline, etc. Getting an example of malware can be problematic since public repositories are quick to remove malicious packages once identified. Consider leveraging ActiveState's repository of Python, Perl, Ruby, etc malware that we've removed from our ActiveState Platform catalog and archived for researchers.
- **Take a "Chaos Monkey" approach** to help flesh out a defense-in-depth strategy using redundant instances and controls. Netflix's Chaos Monkey randomly terminates instances in production to help identify single points of failure.

While these techniques will help ensure your systems and controls are both strong and resilient, attacks are always evolving. Threat modeling should be an ongoing, or at least a periodic practice to ensure your software supply chain remains secure.



Cultural Buy-In

Finally, the last step is often the most difficult, and the most important: getting cultural buy-in. Traditionally, developers, DevOps and other coders are rarely incentivized to emphasize security at the expense of deliverability and features. But without their buy-in, you will always be fighting an uphill battle.

While some frameworks insist that universal buy-in be the starting point of any software supply chain security initiative, we've found that it's always easier to get dev buy-in once you can show them the systems and processes you've put in place won't slow them down.

At ActiveState, we've spent more than twenty years ensuring that the easiest way to work with open source just happens to be the most secure. And now with the ActiveState Platform, organizations can benefit from:

- SLSA Build Level 3-compliant open source runtime environments automatically built from vetted source code in a repeatable manner, along with the attestations to prove it.
- A universal package management tool that simplifies dependency and environment management.

All of which makes it easier for developers to build and use open source, while making it safer for enterprises to adopt.

Conclusions

The journey to a secure software supply chain is just that: a journey, rather than a destination. After all, bad actors will always come up with novel approaches to find and exploit the weakest link in your software supply chain. It's also important to realize that your supply chain is never set in stone:

- Open source authors change
- Packages are constantly updated, become vulnerable, and get patched
- Languages go EOL
- Repositories move
- Trusted vendors change

Our ***Journey to a Secure Software Supply Chain*** is a good overview, but when it comes to implementation, the devil is always in the details. ActiveState's experts can help you understand what supply chain security can mean for your organization, as well as provide advice on the best way to implement it. Feel free to reach out to us at any time on your journey.

References:

Check out all the resources referred to in the book on one handy page.

https://www.activestate.com/journey-to-software-supply-chain-security-re-sources

About the Authors

Dana Crane

With 25+ years in the software industry, Dana has had his share of both crossing and falling into the chasm. He's currently the Product Marketing Manager at ActiveState Software. You can find more of his work at **danacrane.medium.com** and **danacrane.substack.com**

Scott Robertson

Passionate about creating products that solve real problems, Scott drives ActiveState's technology vision based on his experience of over 20 years knees deep in code. Over that time, he's authored a book, founded 3 startups and sold one of them. As ActiveState's CTO, he understands the pains faced in pushing software into production and the challenges big business has to stay fast and relevant. He helps companies do both.

About ActiveState

ActiveState is the de-facto standard for millions of developers around the world who have been using our commercially-backed, secure open source language solutions for over 20 years. Automatically build secure open source language runtime environments (such as Python, Perl, Ruby and more) from source code for Windows, Linux or Mac—all without requiring language or operating system expertise.

www.activestate.com
Toll-free in NA: 1-866.631.4581
solutions@activestate.com