Bootstrapping Trust in Community Repository Projects

Sangat Vaidya[1(✉)], Santiago Torres-Arias[2], Justin Cappos[3], and Reza Curtmola[1]

[1] New Jersey Institute of Technology, Newark, NJ, USA
{ssv33,crix}@njit.edu
[2] Purdue University, West Lafayette, IN, USA
santiagotorres@purdue.edu
[3] New York University, New York, NY, USA
jcappos@nyu.edu

Abstract. Community repositories such as PyPI and NPM are immensely popular and collectively serve more than a billion packages per day. However, existing software certification mechanisms such as code signing, which seeks to provide to end users authenticity and integrity for a piece of software, are not suitable for community repositories and are not used in this context. This is very concerning, given the recent increase in the frequency and variety of attacks against community repositories. In this work, we propose a different approach for certifying the validity of software projects hosted on community repositories. We design and implement a *Software Certification Service (SCS)* that receives certification requests from a project owner for a specific project and then issues a project certificate once the project owner successfully completes a protocol for proving ownership of the project. The proposed certification protocol is inspired from the highly-successful ACME protocol used by Let's Encrypt and can be fully automated on the SCS side. It is, however, fundamentally different in its attack mitigation capabilities and in how ownership is proven. It is also compatible with existing community repositories such as PyPI, RubyGems, or NPM, without requiring changes to these repositories. To support this claim, we instantiate the proposed certification service with several practical deployments.

Keywords: Software certification · Trust establishment

1 Introduction

Community repositories such as PyPI [34], RubyGems [36], and NPM [32] are among the most popular and accessible ways of publishing and distributing open source software. Their immense popularity is illustrated by the large number of downloads: PyPI, the Python package manager, sees more than 600 million downloads per day [35]; npm, the Javascript package manager, more than 700 K downloads a day [33]; RubyGems, the public repository of Ruby packages, has seen more than 107 billion downloads since its creation (as of August 2022).

F. Li et al. (Eds.): SecureComm 2022, LNICST 462, pp. 450–469, 2023.
https://doi.org/10.1007/978-3-031-25538-0_24

Due to their popularity, attacks against community repositories have been on the rise in the recent past [1,3,4,7,10,13,15,16,37]. For instance, in July 2021, a PyPI package containing a backdoor was downloaded almost 30,000 times before the breach was detected [10]. In April 2020, a supply chain attack on RubyGems used packages with names similar to popular packages to infect the end user's system [37]. Similar types of supply chain attacks have become a rising concern for users of NPM as well [1,3,7,13].

This increase in frequency and variety of attacks against community repositories makes it necessary to improve the overall security stance of these popular custodians of open-source software. In this work, we focus on a fundamental question: *How can end users retrieve an authentic version of a community repository project, as intended by the project owner?* Trust in a software project can be bootstrapped by ensuring that what is retrieved is what the project owner intended. When a software project is digitally signed, this question becomes: *How can end users obtain an authentic version of the project owner's public key?*

Looking at existing mechanisms to certify software, we realized that they may not be appropriate in the context of community repositories. While code signing certificates [24–26] ostensibly provide a means to validate the identity of the software publisher, apart from a few large companies, they are rarely used in practice. This sort of certification often requires out-of-band verification and cannot be easily automated. As a result, unfortunately, the effort required to obtain a certificate is prohibitive, making these unsuitable for all types of software projects. We elaborate more in Sect. 3 on the limitations of existing certification mechanisms, including code signing.

In this work, we propose a different approach for certifying the validity of software projects hosted on community repositories. To leverage the existing PKI model of trust, our goal is to provide a way to bootstrap trust using this mechanism. In the PKI model, a certification authority binds a domain owner and a domain name to a public key. The domain owner provides proof of ownership in order to get the X.509 domain certificate. Similarly, we propose a solution where the software project owner proves the ownership of the project and gets a digital certificate that binds the project owner and the project name to a public key. We design and implement a *Software Certification Service (SCS)* that receives certification requests from a project owner for a specific project and then issues a project certificate once the owner successfully completes a procedure for proving project ownership. This project certificate validates a public key for the project.

Unlike in the code signing model, which seeks to establish trust in the identity of the software publisher using a cumbersome procedure, the proposed Software Certification Service relies on a certification protocol with the project publisher to establish ownership of the software. The proposed certification protocol is inspired from the highly-successful ACME protocol [2] used by Let's Encrypt [30] and can be fully automated on the SCS side. It is, however, fundamentally different in its attack mitigation capabilities (*i.e.*, compromise resiliency) and in how ownership is proven (*e.g.*, how to account for the specifics of software naming as

opposed to domain names). It is also compatible with community repositories such as PyPI, RubyGems, NPM, without requiring changes to them.

In the ACME protocol, the owner of a domain proves ownership of that domain by provisioning a specific HTTP resource (*e.g.*, random token chosen by the Let's Encrypt CA) at a specific URL at that domain. In our approach, the project publisher proves ownership over a project by executing a certification protocol with the SCS, which requires the publisher to answer SCS challenges by provisioning certain HTTP resources (e.g., random tokens chosen by the SCS) at a specific location on the project's webpage. The ability to answer SCS challenges proves control over the project's repository. After successfully completing the certification protocol, the project owner gets a project certificate which binds a public key to a (project ID, project owner) tuple. The project owner signs the software project with the corresponding private key for distribution to end users.

The SCS certification protocol includes safeguards to provide resiliency against an adversary who is able to gain control of a project repository (*e.g.* by compromising the project repository credentials). First, the certification protocol is designed to last over an extended period of time. We raise the bar to adversaries who must maintain control over a project for a prolonged period of time, which is arguably more difficult to achieve while going undetected. Second, the SCS protocol requires that the response to a challenge must be placed on the project's repository in a publicly visible way (*i.e.*, the project's webpage). This will prevent an adversary to execute the certification protocol in a stealthy manner.

Finally, we instantiate the proposed service with several practical deployments. First, we use the service to automate the certification of community repositories projects. Our deployments include several popular community repositories: PyPI, RubyGems, and NPM. Second, we use the service to automate the delegation process in community repositories that rely on systems like TUF [12,18] to provide compromise resilience. We are actively working with Google and PyPI on integrating our service into existing cloud security frameworks.

2 Background on the ACME Protocol

The software certification protocol proposed in this paper is modeled after the Automatic Certificate Management Environment (ACME) protocol [2], which can be used by a certificate authority (CA) and an applicant to automate the process of verification and HTTPS certificate issuance. Certificate issuance using ACME resembles a traditional CA's issuance process, in that a user creates an account, requests a certificate for a domain, and proves control of the domain in that certificate in order for the CA to issue the requested certificate.

The entities interacting in the ACME protocol are the ACME client (*i.e.*, the applicant for the HTTPS certificate) and the ACME server (*i.e.*, the CA who issues HTTPS certificates). To begin the process of certificate issuance, the ACME client generates a key pair whose public key will be included in the HTTPS certificate to be generated by the CA. The client proves knowledge of the corresponding private key by signing a CSR (certificate signing request).

The client then engages in a protocol with the ACME server to prove control over the requested domain. For this, the client needs to complete a challenge issued by the ACME server. Once the validation is successful, the client sends a certificate signing request (CSR) just like in the traditional certificate issuance process. On receiving the request, the CA issues the certificate.

The ACME protocol is similar to a traditional certificate issuance protocol. However, the major difference lies in the step where the client proves control over the domain. For a traditional CA, this step requires human intervention. Instead, ACME automates these processes. Let's Encrypt [30] is a free, automated, and open CA which relies on ACME to issue domain certificates. Since its debut in September 2015, it has grown rapidly to become the largest CA on the web.

3 Existing Software Certification Mechanisms

3.1 Code Signing

The code signing model mirrors the PKI model used to issue domain validation TLS X.509 certificates. A software publisher applies for a publisher certificate with a Certificate Authority (CA) and proves its identity in the process. Having verified the publisher's identity, the CA issues a *code signing certificate* which binds the identity of the software publisher to a public key. The publisher then signs the software using the private key corresponding to the public key in the certificate. Finally, the user downloads the signed software, verifies the signature, and validates the publisher's certificate.

Code signing provides the following two guarantees: (1) Validation of the software publisher, *i.e.*, the software comes from a known publisher, and (2) Software integrity (*i.e.*, it has not been modified since it was signed and released by the publisher). The code signing certificate that accompanies the software provides a guarantee that certain checks were done by the CA about the identity of the publisher. As such, it fits best scenarios in which end users need to establish the trustworthiness of the publisher.

Unfortunately, to have its identity verified by the CA, the software publisher needs to go through a very cumbersome process. In addition to verifying that the publisher controls the domain name(s) listed on certificate, the CA need to verify the legal, physical and operational existence of the publisher's business before issuing the certificate. This requires the publisher to provide relevant documents and answer phone calls to complete validation. The CA must also verify the name, title, authority and signature of the person(s) requesting the certificate.

Given this manual and lengthy validation process, code signing certification cannot be automated and imposes large operational costs for the CA. The current code signing model is not suitable for all types of software, as it might be difficult for small businesses, start-ups, independent developers and freelancers to afford a code signing certificate (which few users will validate) that incurs significant costs. Another direct consequence of the cumbersome and intrusive certification process is the low adoption rate for code signing certificates. Besides limited use cases inside closed ecosystems such as Microsoft (for the Windows ecosystem

and MS Office objects), Apple (for software developed using Xcode), and Adobe (for Adobe Air applications), code signing remains largely unused for the large majority of software, including open source software.

3.2 Package Signatures

Some community repositories allow their packages to be cryptographically signed with a private key so that end users can verify the packages with a public key. There are generally two types of package signing:

Signed-By-Repository: In community repositories such as NPM [32], the repository signs uploaded packages with a repository private key. The corresponding public key is publicized on Keybase [29] and is used by end users to verify downloaded packages. The repository private key is kept online to ensure that new packages can be signed as soon as possible. This results in a coarse-grained security guarantee. A compromise of the repository invalidates the security of all its packages. If, on the other hand, the repository private key remains secure, a package signature guarantees that the package uploaded to the repository is the package that an end user downloads. This in itself does not account for the possibility that an individual project's credentials were compromised (even for a brief amount of time) and a malicious version of a package was uploaded to the repository.

Signed-By-Author: In community repositories such as RubyGems [36], a package is signed by its author before being uploaded. The private key used for signing is kept offline. In turn, end users verify the end-to-end authenticity of the downloaded packages based on the corresponding public key. The problem with this model is that end users must discover the correct key by using out-of-band channels, a manual process that is vulnerable to fake key distribution attacks. Alternatively, the authenticity of a public key can be established using a PGP decentralized "web of trust", in which authors vouch for each other's GPG keys.

Some repositories, such as RubyGems, allow the project owner to upload a public key in a dedicated location of the repository – this is a mechanism that can be used to distribute the owner's public key. However, this solution is vulnerable to an attacker that gains control over a project's repository and replaces the owner's authentic public key. The solution we propose provides better resiliency against attackers that gain control over a project's repository.

4 System and Threat Model

4.1 System Model

Figure 1 describes the general architecture of the proposed software certification service. At a high level, our approach is similar to the model employed by code

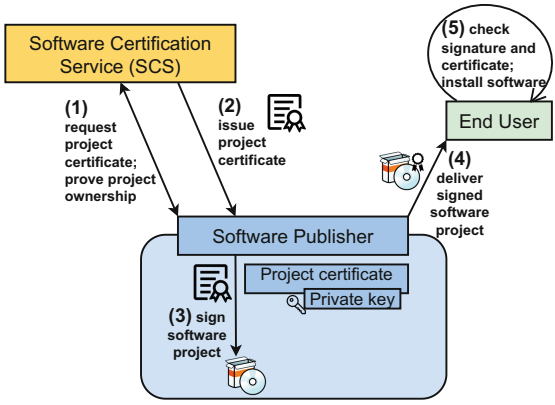


Fig. 1. The software certification architecture.

signing. A *software publisher* contacts the *Software Certification Service (SCS)* requesting a *project certificate* for a software project that it owns (*e.g.*, a Python package hosted on the PyPI community repository). The software publisher then proves ownership of the software project by executing a *software project certification* protocol with the SCS (Step 1). Once the certification protocol is completed successfully, the SCS issues a *project certificate* that binds together a *certificate public key* to a (project ID, project owner) tuple (Step 2). The software publisher then uses the corresponding private key to sign the software project (Step 3) for distribution to end users (Step 4). Finally, end users can verify the integrity of the retrieved software by checking the signature on the software and can get assurance that the software is authentic and originates with the project owner by checking the project certificate (Step 5).

The main difference from the code signing model is in Step 1. Whereas code signing seeks to establish trust in the identity of the software publisher based on a manual procedure that requires human intervention, our approach relies on a certification protocol that requires the software publisher to establish ownership of the software – a protocol designed to be fully automated on the SCS side.

Software publishers that wish to apply for a project certificate need to establish an account with the SCS. This account will be used by the SCS to track interactions with the software publisher. During the software certification protocol, messages sent by a software publisher to the SCS server are authenticated using the publisher's *SCS account key*. Software publishers use a different set of credentials to manage projects hosted on a community repository, referred to as a *repository key* (*e.g.*, a password used to log into the community repository).

Community Repository. We describe the salient features of a *community repository*, which hosts and distributes third party software that represents the main target for the proposed certification service. A community repository is a collection of individual projects which, usually, are open source and are developed

using the same programming language. For example, PyPI [34] (the Python package index), RubyGems [36] (the Ruby package manager), NPM [32] (the JavaScript package manager), or CPAN [27] (the Perl module manager).

Each project has a web-based homepage with a standard format that is uniform across all projects hosted on the same community repository. Typically, a project's homepage contains several sections that can be edited by project owner, such as the project name, owner details, project description, and download links. The proposed certification service leverages the *project description* section of a project's homepage during the protocol used to prove ownership over a project.

4.2 Threat Model and Security Goals

We assume that the SCS service will face adversaries that fit the following threat model. The SCS service (*i.e.*, the SCS server) is trustworthy and the private key used by the SCS service for signing project certificates is out of the attacker's reach. We assume that a software publisher is able to protect her certificate signing key (this is the private key corresponding to the certificate public key). For example, this key can be stored offline, and only be used to sign new project releases. The communication between the SCS server and software publishers (acting as clients) happens over a secure channel (for example using SSL/TLS). We also assume that standard cryptographic primitives can be deployed, such as digital signatures that guarantee integrity and authenticity.
We consider the following types of adversaries:

A1: An attacker who gains access to the client's SCS account. This means that the attacker controls the SCS account key that is used to authenticate a publisher's messages to the SCS server. In this case, the attacker is able to impersonate a software publisher to the SCS service.
A2: An attacker who gains access to the project's repository account. This type of attacker controls the credential used by the project owner to manage the project on the community repository (*e.g.*, a password). This allows the attacker to arbitrarily change content in the project repository, including modifying the project description, adding/deleting project versions, or modifying the project files.
A3: An attacker who executes a network MITM attack between the SCS client and the SCS server. This type of attacker may be a nation state that has the ability to tamper with messages exchanged between the publisher and the SCS service as part of the software certification protocol.

Although an A2-type adversary may gain access to a project's repository account, we assume that the attacker does not control the entire infrastructure of the community repository. As such, the attacker cannot cause the community repository to provide different views of the project repository to different sets of clients. In addition, as our goal is to ensure the security of the certification protocol, we assume that the following attacks are outside the scope of this work:

– An attacker modifies the software package directly in the community repository, or its source code in the corresponding version control system (e.g., a

GitHub repository), and this goes unnoticed by the project owner/maintainer. We assume that proper checks are in place before a community repository project is signed for release.
– Name typosquatting attacks, in which the attacker registers a package with a similar name as a target package.

Attacker Goals: The attacker seeks to obtain a valid signed project certificate that binds a tuple (project ID, project owner) to a public key PK, such that the attacker is not the owner of this project and it possesses the private key corresponding to PK. This will allow the attacker to sign arbitrary versions of the project (*e.g.*, a malicious version that has a backdoor embedded).

Security Goals. Only the legitimate owner of a project should be able to complete a certification protocol for that project. Still, we need to account for occasional events when an attacker gains control over a project's repository, *i.e.*, we need to provide compromise resilience.

Of particular interest are adversaries that can gain control over a project for a short amount of time, during which they may try to obtain a project certificate by executing the certification protocol stealthily. If, on the other hand, adversaries must maintain control over a project for a prolonged period of time in order to successfully complete the certification protocol, this is arguably more difficult to achieve while going undetected. This is especially true if the certification protocol produces artifacts that are publicly visible on the project's webpage.

Concretely, we aim to achieve the following security goals:

SG1: Only an entity that controls an identifier should be able to successfully complete the certification for that identifier (by completing the given challenge). In particular, only the owner of a software project should be able to complete the certification protocol for that project.
SG2: Messages generated during one execution of the certification protocol for one account (i.e., between the SCS server and one client) cannot be used towards obtaining authorizations for other accounts.
SG3: Attackers that gain control over a project's repository for a short period of time should not be able to successfully complete the certification protocol. This prevents such attackers from obtaining a project certificate unbeknownst to the project owner.
SG4: Anyone who can access a project's webpage should be able to know whether an instance of the certification protocol is currently running for that project. In particular, the project owner should be able to tell if someone other than the project owner is trying to obtain a certificate for the project.

5 Software Certification Service

5.1 Preliminaries

General Terms. During the course of execution of the proposed protocol for software certification, we make use of the following terms:

- SCS server: The server software run by the Software Certification Service (SCS) acting as a Certification Authority (CA) that issues project certificates upon request by software publishers.
- SCS client: The client software run by a software publisher that interacts with the SCS server in order to obtain a project certificate for a project owned by that publisher.
- Project Repository: The repository used for hosting the project. This refers to an individual project repository hosted on a community repository.
- Project: The project/package for which the certificate is requested.
- Project Owner: The software publisher who owns the project for which certification is requested. The project owner controls the SCS client and the project hosted on the repository.
- End Users: The users that download the project distribution from the project repository for installation and use.

Keys. The SCS server has a *CA key pair*, and uses the CA private key to sign project certificates. The CA private key has high value and its compromise can have serious consequence for the security of the SCS service. As such, it must be kept offline, or protected using dedicated hardware (e.g., HSMs).

The following types of keys are used by the project owner:

- *SCS account keys* (public/private key pair): Used to authenticate an SCS account holder (acting as a client) to the SCS server. Specifically, the client uses the SCS account private key to sign the messages sent to the SCS server while executing the SCS certification protocol. There is only one SCS account key pair per client, generated by the client. Once registered with the SCS server, an SCS account key can be used to obtain multiple project certificates for multiple projects owned by the client.
- *certificate keys* (public/private key pair): This key pair is generated by the client (acting as a project owner) and its public key is included in the project certificate generated by the SCS. The corresponding private key will be used by the project owner to sign a software project.
- *repository key*: This is the credential used by a project owner to manage the project on the community repository. For example, it can be the password used by the project owner to log into her account with community repositories such as PyPI, RubyGems or NPM.

High-Level Details. As our proposed protocol is inspired from the ACME protocol, we reuse several of ACME's protocol design choices. We mention these details here, so as not to overload unnecessarily the actual protocol description.

JSON Objects and Signatures. Information exchanged between the SCS server and clients is encapsulated in objects encoded as JSON messages [14] carried over HTTPS. Typically, the client sends to the SCS server a stub object, and the server returns the object where various fields have been filled.

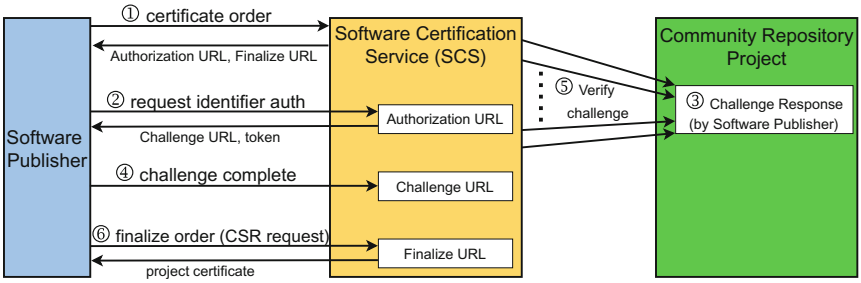


Fig. 2. SCS protocol overview (Phase 2: Obtaining a project certificate).

Messages sent by the client to the server are signed using the private key of the client's SCS account key pair. The server uses the corresponding public key to verify the authenticity and integrity of messages from the client.

Nonces Against Replay Attacks. To ensure protection against replay attacks, the protocol uses an anti-replay mechanism based on *nonces*: The server maintains a list of nonces issued to clients, and any signed request from the client must include a nonce. The server verifies that the nonces it receives from clients are among those that it has issued to clients, and ensures that nonces can be used at most once by clients.

5.2 Certification Protocol Description

We now describe the protocol used by the SCS to issue a software project certificate. The protocol has two major phases: 1) Register an account with the SCS server; 2) Request a project certificate. Phase 1 is carried out only once, when the publisher is communicating with the server for the first time. Each publisher creates an account with the SCS server, so that the SCS server can keep track of its interactions with different publishers. The same account can then be used to get certificates for multiple projects owned by the publisher. Phase 2, illustrated in Fig. 2, is carried out every time the publisher needs a certificate for a project. Appendix A provides a security analysis of the proposed certification protocol.

SCS Account Registration. The protocol execution is initiated by the publisher (*i.e.* project owner) using the SCS client. To register an account with the SCS server, a publisher emgages in the following protocol with the SCS server:

1. The client generates a fresh pair of SCS account keys (public/private keys).
2. The client sends to the SCS server a registration request that contains the following information: the contact details of the client (email address), the SCS account public key, and a signature over the entire registration request using the SCS account private key.

3. The SCS server verifies that the signature is valid and that no account is already registered under this SCS account public key. The server then creates an account and stores the SCS account public key used to verify the registration request. This SCS account key is used to uniquely identify the account and will be used to authenticate future requests from this account.
4. The SCS server informs the client that the account was successfully created.

Obtaining a Project Certificate. To obtain a project certificate, a publisher who has previously registered an SCS account, takes the following four steps:

(1) Submit a project certificate order. The client sends to the SCS server a project certificate order request that contains the software project identifier for which the certificate is requested (*e.g.*, project URL), and the certificate expiration date. Upon receipt of the order request, the SCS server performs some basic checks regarding the project identifier, such as checking the validity of the project URL. The server may also check if the project URL matches one of the participating community repositories.

The SCS server then informs the client that the order is created, together with an "`expires`" time by when the client needs to complete authorization of the requested project identifier. The server's response also contains an Authorization URL (a location on the server where the server makes available an identifier authorization resource associated with this new order request) and a Finalize URL (a location on the server where the client will inform the server that it has completed the project ownership proof requirement).

(2) Obtain authorization over the project identifier. The project identifier authorization process establishes that an SCS account holder is authorized to manage project certificates for a given project identifier. For this, the client must prove ownership over the project by completing multiple validation challenges chosen by the SCS server. To complete a validation challenge, the client provisions the challenge response on the project's repository (more details in Sect. 5.3). The following steps are executed in order to complete a validation challenge:

1. The client sends a request to the Authorization URL and the SCS server responds with an Authorization object that contains the project identifier (*i.e.*, the project URL), the Challenge URL, and a validation token for this challenge. The validation token is a string randomly generated by the SCS server for this challenge. The Challenge URL is a location on the SCS server where the client will notify the server that the challenge has been completed.
2. The client completes the challenge by provisioning the challenge response on the project's repository.
3. The client notifies the SCS server that the challenge was completed by sending a request to the Challenge URL.
4. The SCS server verifies that the challenge was completed.

To address the threat model described in Sect. 4.2, the SCS certification protocol requires a client to respond to multiple challenges spread over time,

and the SCS server to check that the client's response to the challenges remains persistently visible on the project's repository. In Sect. 5.3, we provide details on how challenges are completed by the client and verified by the SCS server.

(3) Finalize the order by submitting a CSR. Once the client completes the server's requirements for this project certificate order, it generates a certificate key pair (public/private keys). It also creates a Certificate Signing Request (CSR) and requests to finalize the order by sending the CSR to the Finalize URL. The CSR contains the software project identifier for which the certificate is requested (e.g., project URL), the certificate public key, the project owner details (name, email address), temporal information (valid from date, expiration date), and a signature over the entire CSR using the certificate private key.

If the request to finalize the order is successful, the SCS server issues the project certificate, which is signed with the server's CA private key. The SCS server then responds to the client with a Certificate URL.

(4) Download the project certificate. The client downloads the project certificate by sending a request to the Certificate URL, located on the SCS server.

5.3 Identifier Authorization

An attacker who gains control over the project's repository for a brief period of time may be able to provision the challenge response on the project's repository, notify the server to validate the challenge, and then quickly remove the challenge response from the project's repository. In order to achieve security goal SG3 and mitigate attackers that can take control of the project repository for a brief period of time, we design the identifier authorization step to last over an extended period of time. In this way, a successful attacker needs to maintain control over the project repository for a longer period of time, which is arguably more difficult to achieve while going undetected.

Specifically, to obtain authorization over a project identifier, a project owner acting as a client in the certification protocol must complete not just one challenge, but multiple validation challenges spread over an *identifier validation window* of time. Additionally, for each challenge, the client must not only provision the challenge response on the project's repository, but must also maintain persistently this challenge response on the project's repository over a *challenge validation window* of time. For example, we may consider a 7-day identifier validation window[1] during which the server will send a new challenge every 24 h for 7 d in a row. For each challenge, the server will check the persistence of the challenge answer on the project's repository multiple times randomly during the 24-hour challenge validation window.

The project identifier authorization process establishes that an SCS account holder is authorized to manage project certificates for a given project identifier.

[1] We picked 7 d based on previous repository breaches, which were detected as early as a few hours in some cases or it took 5–7 d in other cases [21–23].

Validation of individual challenges. For each validation challenge, the client must provision a challenge response on the project's repository. In order to achieve security goal SG4 and deal with long-term adversarial presence, the validation requires that the response to a challenge must be placed on the project's repository in a publicly visible way. This will prevent an adversary to execute the certification protocol stealthily, as the legitimate project owner and/or other project maintainers will notice that a certification protocol is ongoing.

Specifically, to complete a challenge, the client must provision the challenge response in the *project description* section of the project's homepage. To preserve the functionality of the project description section and reduce confusion for the casual user who browses that project's homepage, the challenge response is placed at the end of the project description, using delimiters that make it clear they are not part of the actual project description. Placing the challenge response in the project description meets our requirement that the certification protocol must generate artifacts that are publicly visible.

The client generates the challenge response as a Base64-encoded string of characters generated by concatenating the validation token for the challenge with a key fingerprint, separated by a"." character:
Response = token || "." || base64(fingerprint(SCS account key)),

where "||" denotes concatenation of strings, and the fingerprint is computed as a SHA-256 digest of the SCS account key. The response is placed at the end of the project's description, using clear delimiters.

After notifying the server about completion of the challenge, the client needs to maintain the challenge response on the project homepage during the challenge validation window. The server checks the existence of the challenge response multiple times at random times within this window. If all the server checks during the challenge validation window are successful, the server deems the challenge as successfully completed, and generates the next challenge for the client.

6 Deployments

6.1 SCS Implementation Details

The SCS service has two components, the server and the client. We implemented the SCS server on top of Boulder [39], which is an open-source ACME-based CA built for Let's Encrypt and written in Go. We have adapted the code to process the Project ID (the project repository URL) instead of the domain names. For example, when the client requests a project certificate, the server verifies that the project URL comprises of a valid set of characters and that the URL belongs to one of the community repositories that the SCS service has been deployed to. We also implement the challenge-response protocol used for proving project ownership. The SCS server is engages by keeping track of the challenge-response process and how far the client is in the proof of ownership process. The process on the server side is automated and does not require manual intervention.

We implemented the SCS client on top of Lego [31], which is an ACME client implementation for Let's Encrypt, written in Go. The SCS client is responsible

for initiating the certificate issuance process, by placing a request to the server. The client is also responsible for participating in the challenge-response protocol and for fulfilling the challenge issued by the server. The project owner gets the challenge response from the SCS client and provisions it on the project homepage. This is the only step that requires manual intervention during the challenge-response SCS protocol execution.

6.2 Deployment to Community Repositories

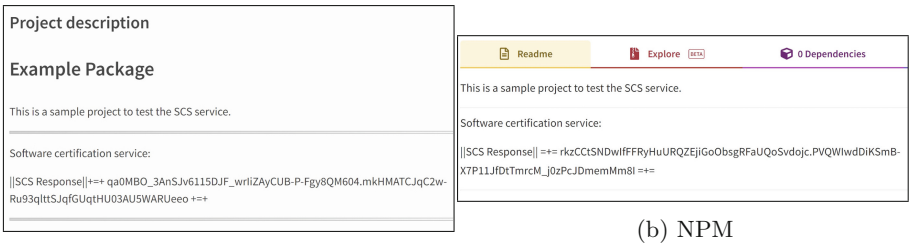
We deployed the SCS service to several community repositories to automate the issuance of certificates for the projects hosted on these repositories. By design, the SCS service does not require any changes to these community repositories, which makes it deployable right away and serves as an incentive for adoption.

The SCS service can be deployed to community repositories where each individual project has a dedicated webpage containing a project description section. Most community repositories fit this scenario, with the project description being normally used to provide basic information about the project. Although every community repository may have a different web layout for the project description, all the projects that are hosted on the same community repository have the same layout for the project description.

During the SCS protocol execution, the project owner provisions on the project description webpage the responses to challenges issued by the SCS server. To preserve the functionality of the project description field and reduce confusion for the casual user who browses the project's webpage, the challenge responses are placed at the end of the project description, using delimiters that make it clear they are not part of the actual project description (see Sect. 5.3). As shown in Fig. 3, the challenge response will be publicly visible on the project's webpage.

SCS for PyPI. PyPI [34] is used for hosting and distributing Python packages. We use the "Project description" page to display the SCS challenge response. For this, the project owner includes the challenge response in the project description section of the *setup.py* file, which generally contains the metadata for the Python package, and then builds the package and uploads it to PyPI.

SCS for RubyGems. RubyGems [36] is used for hosting and distributing Ruby projects, known as "gems". To display the SCS challenge response, we use a section on a project's webpage where the owner can provide a short description of the project, which can range from a single sentence to a few paragraphs. Also, the project page does not allow HTML or Markdown formatting and so, unlike in PyPI, project owners do not have any choice in the way a challenge response gets displayed in the section. The project owner includes the challenge response in the description field of the *gemspec* file, which generally contains the metadata for the gem, and then builds the package and uploads it to RubyGems.



(a) PyPI

(b) NPM

Fig. 3. Challenge response on the project webpage for various community repos.

SCS for NPM. NPM [32] is the Node package manager used for hosting and distributing JavaScript packages. We use the "Readme" page to display the SCS challenge response. For this, the project owner includes the challenge response in the *package.json* file and then builds the package and uploads it to NPM.

6.3 Automating Delegations in Community Repositories

We consider a setting in which a system such as TUF [18] or Diplomat [12] is used to provide compromise resilience for a community repository such as PyPI. This type of protection is achieved through several mechanisms, such as the use of *roles* (which allow to separate responsibility in a system) and *delegations* (which allow to distribute responsibilities in a system).

For PyPI, there is a `root` role, which indicates which keys are authorized for other roles, such as the `projects`, `release`, and `timestamp` roles. The `projects` role is trusted to validate all the packages on PyPI. This role delegates trust for individual packages to the developers responsible for those packages. For example, the `projects` role may delegate the `BeautifulSoup` project to the public key belonging to the developer Alice, who is responsible for `BeautifulSoup`.

This delegation step can occur whenever a new project is created on PyPI, or an existing project wants to change an existing delegation. Currently, such a delegation involves manual operations on the part of the PyPI maintainers, which is not scalable since PyPI has over 345,000 projects (as of December 2021).

We automate this delegation step using the SCS service. To have her public key certified as trusted for the `BeautifulSoup` project, the developer responsible for `BeautifulSoup` engages in the SCS ownership-proving protocol with the entity responsible for the `projects` role (i.e., the PyPI server). If the developer successfully completes the SCS protocol, this serves as proof that the developer owns the `BeautifulSoup` project. As a result, the `projects` role will delegate trust for the `BeautifulSoup` project to the public key of this developer.

Specifically, once the ownership protocol is completed successfully, the server updates the top-level `projects` role to include a new "delegations" entry to a new role `BeautifulSoupOwner` that is in charge of `BeautifulSoup`. This entry will include the public key of the developer responsible for `BeautifulSoup`. Then, the developer creates the `projects` file for the `BeautifulSoupOwner` role.

7 Related Work

Previous works in the area of securing community repositories studied the design and implementation of community repositories and proposed attacks [5,6] and defenses [11,12,17]. These works focus on designing more secure software ecosystems with properties such as compromise-resilience and supply chain integrity. [19] discusses the security issues with the programming language specific community repositories like PyPI, RubyGems or NPM. In addition, due to the rising number of vulnerabilities and malware in the NPM ecosystem, various works [8,9,40] have been proposed to find new vulnerabilities, isolate untrusted packages, evaluate risks and remediate issues. [20] discusses the typosquatting and combosquatting attacks on the Python software ecosystems like PyPI. Other frameworks, such as in-toto [17,28] and Sigstore [38], focus on the security of the entire software supply chain. As opposed to previous work, our focus is on bootstrapping trust in a community repository project by ensuring that end users can retrieve an authentic version of a community repository project, as intended by the project owner. Specifically, we propose a new mechanism to certify software hosted in community repositories.

8 Conclusion

In this work, we have presented a new approach for certifying the validity of software projects hosted on community repositories. Towards this goal, we have introduced a Software Certification Service (SCS) which gives software publishers the ability to prove the ownership of their projects and then get a project certificate that binds the project owner and the project name to a public key. Although inspired from the ACME protocol in that it can be fully automated on the SCS side, the proposed certification protocol is fundamentally different in its attack mitigation capabilities and in how ownership is proven.

We deployed the SCS service to several community repositories, including PyPI, RubyGems, and NPM, to automate the issuance of certificates for projects hosted on these repositories. By design, the SCS service does not require any changes to these community repositories, which makes it deployable right away and serves as an incentive for adoption. We are currently working with Google and PyPI on integrating our service into existing cloud security frameworks. As future work, we plan to extend the SCS service to more community repositories (currently, we require that each individual project has a dedicated webpage containing a project description section) and to explore other use cases that can benefit from automated verification. We also plan to evaluate the usability aspects of the proposed SCS certification protocol; in particular, we need to better understand what are appropriate values for the validation windows, which should be chosen as a tradeoff between usability and security.

Acknowledgments. This research was supported by the US National Science Foundation under Grants No. CNS 1801430, DGE 1565478, and DGE 2043104.

A Security Analysis

We now turn to analyzing the security of the proposed SCS protocol. We first show that the protocol meets the security goals stated in Sect. 4.2, and then analyze the protocol's compromise resiliency.

SG1: *Only a project's owner should be able to complete the certification for that project.* To prove ownership over a project, which is required for completing the certification protocol, an entity must successfully complete the challenges generated by the SCS server. As such, for each challenge, the entity must both:

- Hold the private key of the SCS account key pair used to respond to the challenge. This is because the responses from the client to the SCS server must be signed with that key.
- Control the project in question. This is because successfully provisioning the challenge response on the project's homepage requires write-access to the project's repository.

Since only the project owner has write-access to the project's repository, a successful execution of the SCS protocol ensures that a specific SCS account holder is also the entity that controls a project (*i.e.*, the project owner).

SG2: *Messages generated during one execution of the certification protocol for one account (i.e., between the SCS server and one client) cannot be used towards obtaining authorizations for other accounts.* This is achieved because all messages sent by an SCS client to the SCS server are signed using that client's SCS account private key. Thus, such messages cannot be reused between instances of the certification protocol executed by different SCS account holders.

SG3: Attackers that gain control over a project's repository for a short period of time are not be able to successfully complete the SCS certification protocol. The certification protocol is designed so that the identifier authorization step lasts over an extended period of time. An entity attempting to complete the certification protocol for a project must complete multiple challenges. For each challenge, the challenge response must be maintained persistently on the project's homepage, because the SCS server will check multiple times randomly during the challenge validation window. If an attacker is able to briefly gain control over the project's repository, she maybe able to provision a valid challenge response for that challenge. However, such an attacker will not be able to successfully provision valid information for subsequent challenges.

SG4: We need to show that an attacker cannot complete an SCS certification for a project in a stealthy manner. The SCS protocol achieves this by requiring that all challenge responses must be placed on the project's repository in a publicly visible way (*i.e.*, on the project's homepage). This ensures that the legitimate project owner and/or other project maintainers will notice that a certification protocol is ongoing.

Compromise Resiliency. If an attacker is able to get hold of the *repository key* for a project, this allows the attacker unfettered access to the project repository, including making changes to the project's homepage. The attacker can register an account with the SCS server and then request a project certificate under this SCS account. Having access to the repository key, the attacker will be able to provision challenge responses on the project homepage.

The SCS protocol has two safeguards in place to deal with a repository key compromise. First, the certification protocol is designed to to last over an extended period of time. Thus, if the repository key compromise is detected early enough, the project owner can change the repository key, preventing the attacker from successfully completing the certification protocol. In this way, a successful attacker would have to maintain control over the project repository for a longer period of time, which is arguably more difficult to achieve while going undetected. Second, the SCS protocol requires that the response to a challenge must be placed on the project's repository in a publicly visible way (*i.e.*, the project's homepage). This will prevent an adversary to execute the certification protocol stealthily, as the legitimate project owner and/or other project maintainers will notice that a certification protocol is ongoing and will take steps to terminate such an active threat.

References

1. Aguirre, J.: Fake npm Roblox API Package Installs Ransomware and has a Spooky Surprise. https://blog.sonatype.com/fake-npm-roblox-api-package-installs-ransomware-spooky-surprise (2021)
2. Barnes, R., Hoffman-Andrews, J., McCarney, D., Kasten, J.: Automatic Certificate Management Environment (ACME). RFC 8555 (Mar 2019). https://datatracker.ietf.org/doc/html/rfc8555
3. Barsan, A.: Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610/ (February 2021)
4. Burt, J.: Supply Chain Flaws Found in Python Package Repository. https://www.esecurityplanet.com/threats/supply-chain-flaws-found-in-python-package-repository/ (August 2021)
5. Cappos, J., Samuel, J., Baker, S., Hartman, J.H.: A look in the mirror: Attacks on package managers. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 565–574. CCS '08, ACM, New York, NY, USA (2008)
6. Cappos, J., Samuel, J., Baker, S., Hartman, J.H.: Package Management Security. Tech. rep., University of Arizona (2008)
7. Cimpanu, C.: Malware found in npm package with millions of weekly downloads. https://therecord.media/malware-found-in-npm-package-with-millions-of-weekly-downloads/ (October 2021)
8. Decan, A., Mens, T., Constantinou, E.: On the impact of security vulnerabilities in the npm package dependency network. In: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 181–191. MSR '18, ACM (2018)

9. Garrett, K., Ferreira, G., Jia, L., Sunshine, J., Kästner, C.: Detecting suspicious package updates. In: Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, pp. 13–16. ICSE-NIER '19, IEEE Press (2019). https://doi.org/10.1109/ICSE-NIER.2019.00012
10. Goodin, D.: Software downloaded 30,000 times from PyPI ransacked developers' machines. https://arstechnica.com/gadgets/2021/07/malicious-pypi-packages-caught-stealing-developer-data-and-injecting-code/ (July 2021)
11. Kuppusamy, T.K., Diaz, V., Cappos, J.: Mercury: Bandwidth-effective prevention of rollback attacks against community repositories. In: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference, pp. 673–688. USENIX ATC '17 (2017)
12. Kuppusamy, T.K., Torres-Arias, S., Diaz, V., Cappos, J.: Diplomat: Using delegations to protect community repositories. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), pp. 567–581 (2016)
13. Lakshmanan, R.: Two NPM Packages With 22 Million Weekly Downloads Found Backdoored. https://thehackernews.com/2021/11/two-npm-packages-with-22-million-weekly.html (November 2021)
14. Rfc 8259. https://datatracker.ietf.org/doc/html/rfc8259
15. Ruohonen, J., Hjerppe, K., Rindell, K.: A Large-Scale Security-Oriented Static Analysis of Python Packages in PyPI. In: Proceedings of the 18th International Conference on Privacy, Security and Trust (PST). IEEE (2021)
16. Sharma, A.: Sonatype Catches New PyPI Cryptomining Malware. https://blog.sonatype.com/sonatype-catches-new-pypi-cryptomining-malware-via-automated-detection/ (June 2021)
17. Torres-Arias, S., Afzali, H., Kuppusamy, T.K., Curtmola, R., Cappos, J.: In-toto: Providing farm-to-table guarantees for bits and bytes. In: Proceedings of the 28th USENIX Conference on Security Symposium, pp. 1393–1410. SEC'19 (2019)
18. TUF: The Update Framework. https://www.updateframework.com/
19. Vaidya, R.K., Carli, L.D., Davidson, D., Rastogi, V.: Security issues in language-based sofware ecosystems. CoRR abs/1903.02613 (2019)
20. Vu, D.L., Pashchenko, I., Massacci, F., Plate, H., Sabetta, A.: Typosquatting and combosquatting attacks on the python ecosystem. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS PW). pp. 509–514 (2020). https://doi.org/10.1109/EuroSPW51379.2020.00074
21. Bitcoin gold issues critical alert. https://www.enterprisetimes.co.uk/2017/11/27/bitcoin-gold-issues-critical-alert
22. Npm packages disguised as roblox api code caught carrying ransomware. https://www.theregister.com/2021/10/27/npm_roblox_ransomware/
23. Typosquatting attacks on rubygems. https://thehackernews.com/2020/04/rubygem-typosquatting-malware.html
24. Introduction to Code Signing. https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms537361(v=vs.85)
25. Minimum Requirements for the Issuance and Mgmt. of Publicly-Trusted Code Signing Certificates. https://casecurity.org/wp-content/uploads/2016/09/Minimum-requirements-for-the-Issuance-and-Management-of-code-signing.pdf
26. Leading Certificate Authorities and Microsoft Introduce New Standards to Protect Consumers Online. https://casecurity.org/2016/12/08/leading-certificate-authorities-and-microsoft-introduce-new-standards-to-protect-consumers-online/
27. Comprehensive Perl Archive Network. https://www.cpan.org/

28. in-toto. https://in-toto.io/
29. Keybase. https://keybase.io/
30. Let's Encrypt. https://letsencrypt.org/
31. ACME client implementation. https://letsencrypt.org/docs/client-options/
32. Javascript Node package manager. https://npmjs.com
33. NPM download stats. https://npmcharts.com/
34. Python Packaging Index. https://pypi.org
35. PyPI download stats. https://pypistats.org/packages/__all__
36. RubyGems statistics. https://rubygems.org/stats
37. Supply-chain attack hits RubyGems repository with 725 malicious packages. https://arstechnica.com/information-technology/2020/04/725-bitcoin-stealing-apps-snuck-into-ruby-repository/ (2020)
38. Sigstore. https://www.sigstore.dev/
39. ACME server Boulder. https://github.com/letsencrypt/boulder
40. Zimmermann, M., Staicu, C.A., Tenny, C., Pradel, M.: Small world with high risks: A study of security threats in the npm ecosystem. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 995–1010 (2019)