DEVGUARD
BY L3MONTREE

RESEARCH & DEVELOPMENT WHITEPAPER

Exchanging VEX Rules between DevGuard Instances via a Central Exchange Server

Refaei Shikho
refaei.shikho@l3montree.com

Reviewed by
Sebastian Kawelke

L3montree GmbH
Bonn, 2026-11-24

Contents

1 Introduction 1
1.1 Previous Work 1
1.2 Motivation 1
2 Scope 2
2.1 DevGuard and VEX Rules 2
2.2 Crowdsourced Vexing 3
2.3 Exchange Architecture 3
2.4 Workflow 4
3 System Analysis 5
3.1 Transferability of the CV Algorithm 5
3.2 Aggregation within an Instance 6
3.3 Alternative Approaches Revisited 7
4 System Definition 7
4.1 Threat Model 7
4.2 Submission and Consistency Check 8
4.3 Voting on the Exchange Server 10
4.4 Dynamic Trust Score 11
5 Implementation 12
5.1 Registration and Request Signing 12
5.2 Synchronization Daemons 13
5.3 Processing on the Exchange Server 13
6 Evaluation 14
7 Future Work 15
7.1 Limiting the Influence of Unapproved Instances 15
7.2 Costly Registration 15
References 16

List of abbreviations

BFT Byzantine Fault Tolerant

BGP Byzantine Generals Problem

CEL Common Expression Language

CV Crowdsourced Vexing

CVE Common Vulnerabilities and Exposures

DNS Domain Name System

ECDSA Elliptic Curve Digital Signature Algorithm

HTTP Hypertext Transfer Protocol

PoC Proof of Concept

PoS Proof-of-Stake

TLS Transport Layer Security

VEX Vulnerability Exploitability eXchange

1 Introduction

As software projects grow in complexity, security vulnerabilities and bugs are bound to find their way into the codebase and infrastructure, be it through package dependencies or human error [2], [3]. Modern industry increasingly shifts the responsibility for managing these issues from dedicated security experts to software engineers, who are often not equipped to handle this task properly [4], [5], [7]. To support developers in this regard, the open-source DevGuard project, developed and maintained by L3montree GmbH, provides tooling to detect and handle security issues in applications during development [11], [12].

A central part of this workflow is vexing, i.e. the context-aware assessment of whether a vulnerability in a dependency actually affects a project, documented in the Vulnerability Exploitability eXchange (VEX) format [6]. In DevGuard, such assessments can be codified as VEX rules, which bind a condition on the Common Vulnerabilities and Exposures (CVE) and the dependency path to an outcome, e.g. marking a finding as false positive [9]. The dependency path is the chain of dependencies through which a vulnerable package is included in a project, starting at the project itself, via its direct dependencies, down to the affected package. Projects that include a vulnerable package via the same dependency path often face the same questions during the assessment and are therefore likely to arrive at the same result, although the assessment can still differ, e.g. depending on how a project uses the package. Sharing VEX rules can therefore remove a considerable amount of redundant, manual triage work.

1.1 Previous Work

Building on this observation, Quach designed and implemented Crowdsourced Vexing (CV), a system that aggregates the VEX rules of all organizations through trust-weighted majority voting and recommends the most plausible rule for a given CVE and dependency path [13]. The system was derived from a threat model based on attack trees, implemented as a Proof of Concept (PoC) in the DevGuard backend and evaluated with a dedicated test suite. However, the approach explicitly relies on DevGuard's centralized architecture: all organizations, VEX rules and trust scores reside in the database of a single DevGuard instance, and trust scores are assigned by the administrators of that instance [13]. Decentralized concepts such as proof-of-stake or Byzantine fault tolerant consensus were dismissed in that work, mainly because they did not fit a single centralized system.

1.2 Motivation

In practice, this assumption does not hold. DevGuard is open source and can be self-hosted [8], so organizations with strict confidentiality or sovereignty requirements operate their own instances instead of using a shared one. Each of these instances only knows the VEX rules created by its own organizations. As a consequence, the knowledge collected in one instance remains isolated,

and maintainers in other instances have to repeat the very same assessments for the same CVE and dependency paths. The more the user base is distributed across instances, the less data is available to the CV algorithm of each individual instance, which intensifies the cold-start problem and makes minimum voter thresholds harder to reach.

It is therefore desirable to exchange VEX rules not only within, but also between DevGuard instances. This work proposes a central exchange server for this purpose: participating instances submit their VEX rules to the server, which performs the voting across all instances, and in turn every instance can retrieve the resulting rules. The server thus acts as a shared knowledge base across instance boundaries without requiring the instances to communicate with each other directly.

This setting, however, raises questions that did not arise within a single instance. The server has no direct insight into the users and organizations of the submitting instances, so it can neither verify their identity nor their age, and a single instance could report an arbitrary number of fabricated organizations. Trust scores assigned by the administrator of one instance are not comparable to those of another instance and can no longer be assigned centrally for all participants. Furthermore, the communication between instances and the server becomes an additional attack surface, and the submitted rules must not reveal more about the organizations than necessary.

This work therefore investigates how VEX rules can be exchanged between multiple DevGuard instances via a central exchange server. It builds on the previous work [13], references the parts that remain unchanged and focuses on the voting and the threat model, which have to be adapted to instances as voters.

2 Scope

This section briefly summarizes the concepts adopted from the previous work and defines the architecture and workflow considered within this work. For a detailed description of DevGuard, VEX and the CV algorithm, refer to [13].

2.1 DevGuard and VEX Rules

DevGuard continuously scans the dependencies of software projects, links detected CVEs to the affected dependency path and supports maintainers in assessing and mitigating them [8]. The outcome of such an assessment, referred to as vexing, can be documented according to the VEX standard, which states whether a product is actually affected by a vulnerability [6].

To avoid repeating the same assessment, DevGuard provides VEX rules. In the previous work, a VEX rule is described as a CVE identifier, a dependency path pattern that may contain wildcards, e.g. $* \to A \to B$, and an assessment such as `false-positive` or `affected` [13]. In the subsequent

implementation, the condition of a rule was generalized to a Common Expression Language (CEL) expression, which is evaluated against the dependency vulnerabilities of a project [11]. A path pattern is converted into such an expression, e.g. for CVE-2024-45590 in the package `body-parser`, included via `express`:

```
vuln.cveId == "CVE-2024-45590" && matchesPattern(vuln,
   ["*", "pkg:npm/express@4.18.2", "pkg:npm/body-parser@1.20.1"])
```

This expression matches every finding of this CVE whose dependency path ends with `body-parser` included via `express`, regardless of the dependencies before them. If the expression restricts the rule to a single CVE, this CVE is additionally stored as the optional CVE scope of the rule. A rule thus consists of a CEL expression, an optional CVE scope and an assessment. It documents an assessment once, so that it can be reused for all findings that match the expression, instead of assessing each finding separately.

2.2 Crowdsourced Vexing

CV recommends a VEX rule for a given CVE and dependency path by treating every matching VEX rule stored in the instance as a vote of the organization that created it [13]. Votes of organizations younger than 30 days are discarded, each combination of organization, project and asset is counted only once per candidate, and each vote is weighted by the higher of the trust scores of the organization and the project, which are assigned by an administrator. Repeated votes through organizations of the same creator are weighted with a decreasing factor. A candidate is a combination of path pattern and assessment, and the candidate with the highest total weight is recommended if at least four votes exist and no tie occurs.

In the implementation, the matching was adapted to the CEL expressions [11]: the expression of every rule is evaluated against the open dependency vulnerabilities, the matching rules are grouped per dependency vulnerability, and the voting is performed for each dependency vulnerability. Its result is a recommendation together with a confidence value.

The previous work identified a self-reinforcing feedback loop: a rule that a user adopts from a recommendation is counted as a new vote in later aggregations and thus strengthens the majority that produced the recommendation [13]. This was subsequently solved in the implementation of the previous work: each VEX rule carries a flag indicating whether it originated from a recommendation, and such rules are ignored in the voting [11].

2.3 Exchange Architecture

The system considered in this work consists of multiple independently operated DevGuard instances and one central exchange server, as illustrated in Figure 1. Each instance manages its own

users, organizations, projects and VEX rules. The instances submit their VEX rules themselves, not the recommendations computed by their local CV algorithm; before submission, an instance only excludes rules adopted from recommendations and ensures that it sends at most one rule per CEL expression; this consistency check is described in Section 4.2. The exchange server stores the received rules, performs the voting with each instance counting as one voter, and provides the resulting rules together with a confidence value to all instances; the voting is described in Section 4.3. Within each instance, these rules enter the local CV voting alongside the local rules, weighted by their confidence value. Instances communicate exclusively with the exchange server and never directly with each other.

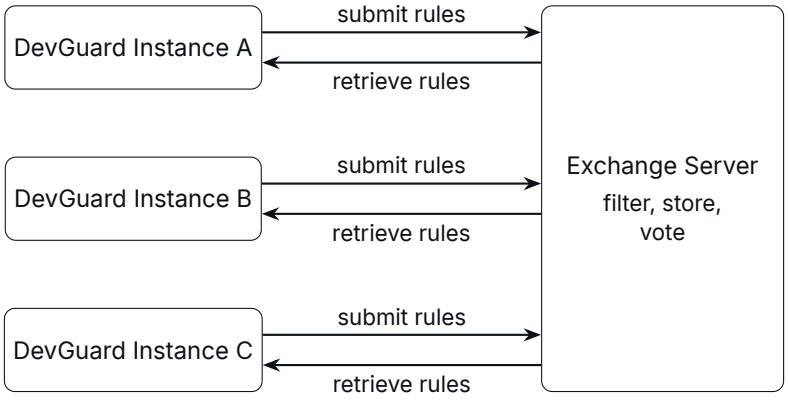

Figure 1. Exchange of VEX rules between DevGuard instances via a central exchange server

Each instance owns a key pair and signs every request with its private key. The exchange server stores the public key of every registered instance together with a status, which the administrator of the exchange server sets manually, e.g. to approve an instance after talking to its operator or to block its key. The server accepts every request whose signature can be verified and whose key is not blocked; the approval increases the trust score of the instance (Section 4.4). As in the previous work, the voting is performed by a central component. In contrast to it, however, the exchange server has no access to the databases of the instances and only knows what the instances submit.

2.4 Workflow

The workflow covered by this work extends the recommendation workflow of the previous work [13] by the exchange between instances and the server. It can be summarized in the following steps:

1. **Registration (once):** The instance registers its public key. Optionally, the server administrator approves it after talking to its operator (*).
2. **Consistency check:** Before submitting, an instance excludes rules that originated from a recommendation, normalizes the CEL expressions of the remaining rules and checks them for contradicting assessments for the same CEL expression (Section 4.2).

3. **Submission:** Routinely, the daemon job of the instance submits all its VEX rules that passed the consistency check in a signed request.
4. **Processing:** The exchange server verifies the signature of the request, validates the received rules, repeats the consistency check per instance and CEL expression, and replaces the previously stored rules of the instance with the accepted ones.
5. **Voting:** Independently of the submissions, a daemon job on the exchange server routinely applies the CV algorithm to the stored rules, with each instance acting as one voter, and stores the resulting VEX rule for each CEL expression together with a confidence value in a separate table.
6. **Retrieval:** The daemon job of the instance fetches the rules and confidence values resulting from the latest voting on the exchange server.
7. **Application:** The retrieved rules enter the local voting of the instance together with its own rules, weighted by their confidence value. The resulting recommendations are proposed to the user, who accepts or rejects them.

Steps marked with (*) are optional. Anything not covered by these steps, such as the vexing process itself or the presentation in the user interface, is considered out of scope.

3 System Analysis

The CV algorithm of the previous work was designed for a single instance in which all entities and their metadata are directly accessible from one database [13]. In the multi-instance setting, the exchange server only receives what the instances report, and only the instance itself can be authenticated. This section analyzes which parts of the algorithm can be transferred to this setting, how the aggregation within an instance affects the voting, and whether the alternative approaches dismissed in the previous work become applicable now.

3.1 Transferability of the CV Algorithm

Each component of the CV algorithm, including its in-scope mitigations, depends on certain information. Analogous to the threat model in Section 4, a component remains *unchanged* if this information is still available to the exchange server, and is *transferred* if it now has to refer to instances instead of organizations.

Components that only operate on the content of a rule remain unchanged, since the exchange server receives the content of every rule. This applies to the validation of the assessment against a fixed set of values (mitigation 30) and the deterministic tie-breaking (mitigation 31). The matching of rules against dependency vulnerabilities is not needed on the exchange server, since it groups

the votes by identical normalized CEL expression. It remains unchanged within the instances, where the retrieved rules are evaluated against the local dependency vulnerabilities as part of the local voting.

All other components rely on metadata about organizations, projects and their creators. In the multi-instance setting, this metadata is generated by the submitting instance and cannot be verified by the exchange server. A malicious instance could report arbitrarily old organizations, distinct creators for every organization or fabricated trust scores, and thereby bypass mitigations 8, 10, 11 and 13 entirely. Consequently, these components have to be based on information the exchange server can establish itself. The only such entity is the instance: it is identified by its key pair, the server knows when it was registered, and its trust score can be derived from its approval and its voting outcomes. This is why each instance, rather than each organization, is treated as one voter. Accordingly, deduplication (mitigation 20) ensures one vote per instance and candidate, the minimum age (mitigations 10 and 11) refers to the registration of the instance, and the trust score (mitigation 13) is assigned per instance.

The diminishing factor (mitigation 8) addressed a single user creating many organizations. Its counterpart in the multi-instance setting is a single operator registering many instances to gain additional votes, which corresponds to a profile injection attack on instance level. Since every instance with a valid key can participate, such an attack remains possible. However, the operator of an unapproved instance is unknown to the server, so the diminishing factor cannot be applied per operator. Instead, unapproved instances only receive a low trust score, and only the manual approval by the administrator of the exchange server raises it. Finally, the minimum voter threshold (mitigation 15) has to count instances instead of organizations. Since the number of instances is considerably smaller than the number of organizations, the threshold has to be chosen carefully to avoid that recommendations are hardly ever issued.

3.2 Aggregation within an Instance

Treating each instance as one voter implies that the assessments of all organizations within an instance are reduced to a single vote per CEL expression. This reduction is performed by the consistency check (Section 4.2).

This reduction has two consequences. On the one hand, it protects the voting against large instances. An instance with many organizations, or with many fabricated organizations, cannot outvote several smaller instances, since every instance contributes at most one vote per candidate. Moreover, an instance that cannot reach an internal agreement does not contribute an uncertain assessment to the voting. On the other hand, information is lost: a unanimous assessment of a hundred organizations carries the same weight as the assessment of a single organization. Differences in the size and reliability of instances can therefore only be reflected through the trust score of the instance. The dominance threshold additionally implies that minority assessments

within an instance are not visible on the exchange server, even if they are correct for the specific context of an organization.

3.3 Alternative Approaches Revisited

The previous work dismissed Proof-of-Stake (PoS) and Byzantine Fault Tolerant (BFT) consensus based on the Byzantine Generals Problem (BGP), mainly because DevGuard is a centralized system and trust scores have no economic value [13]. This result still holds for the multi-instance setting. BFT algorithms allow participants to agree on a common value without a trusted party, provided that less than a third of a fixed set of participants is faulty [1]. Here, the instances do not need to agree among themselves, since the exchange server aggregates their votes as a trusted central party; a compromised server remains a risk (Section 6). Moreover, the set of voters is open, since every instance can register a new key, so an adversary could exceed the bound of faulty participants by registering additional instances. Trust scores still have no economic value that could serve as a stake [10]. Therefore, this work extends the CV algorithm of the previous work, whose transferred components are defined in the following section.

4 System Definition

This section defines the multi-instance system. It re-evaluates the threat model of the previous work for instances as voters and then describes the consistency check, the voting on the exchange server and the dynamic trust score.

4.1 Threat Model

The attack tree of the previous work targets a voting in which organizations and their users are the voters [13]. Since the voters are now instances, the attack tree is re-evaluated. The root goal remains the manipulation of the voting, and the three branches remain the same: compromising voters, manipulating the communication and compromising the algorithm. Their attacks, however, now target instances and the exchange server instead of users and organizations. Figure 2 shows the resulting attack tree, where the numbers refer to the mitigations in Table 1.

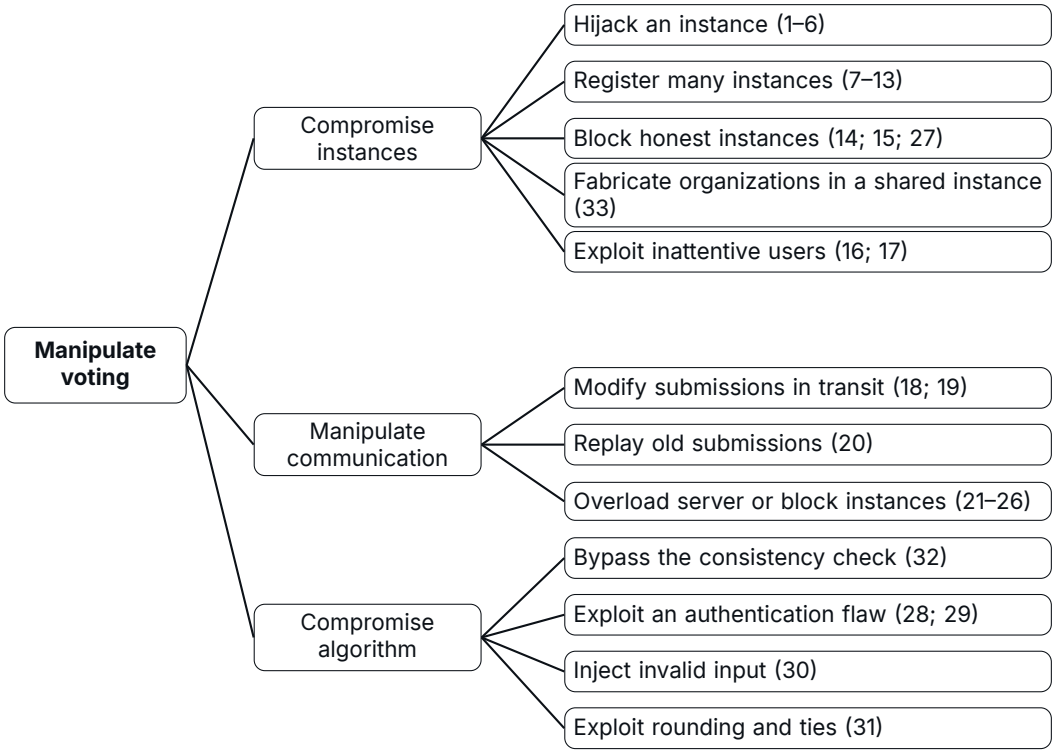

Figure 2. Attack tree for the voting with instances as voters

Table 1 re-evaluates the mitigations of the previous work. Mitigations are *unchanged* if they still apply in the same way, and *transferred* if they now refer to instances instead of users or organizations. Mitigations 32 to 34 are new.

Most mitigations are transferred: attacks on users and organizations now take place within an instance, and each instance is responsible for protecting itself against them. For the exchange server, only attacks against instances as a whole are relevant. A user can still set up several instances to gain additional votes. The only countermeasure against this is that unapproved instances receive a lower trust score than approved ones (mitigations 7 to 9, see Section 3.1). Two new threats arise from the instance as voter: an instance may submit contradicting rules, which the server prevents by repeating the consistency check (mitigation 32), and an adversary may fabricate organizations in a shared instance to tip its consistency check, which can only be limited by decreasing the dynamic trust score of instances that lose a voting (mitigation 33). Finally, mitigation 34 limits the information that leaves an instance.

4.2 Submission and Consistency Check

An instance submits its VEX rules to the exchange server. Each rule only contains its own content, i.e. the CEL expression with its optional CVE scope, the assessment and the title and justification of the rule, but no information about the organizations, projects or assets it originates from (mitigation 34). Since the title and the justification are free text, users are informed when creating a VEX

ID	Previous work	Multi-instance	Status	In Scope
1, 2	Two-factor authentication, token expiry	Key pair per instance, signed requests	Transferred	No
3, 4	System standards, guidelines	Guidelines for instance operators	Transferred	No
5, 6	Account recovery and lockdown	Block the compromised key, register a new key	Transferred	No
7–9	Limit organizations per user, decrease votes of the same user, CAPTCHA	Low trust score of unapproved instances	Transferred	Yes
10, 11	Minimum organization age, age-based trust	Minimum instance age	Transferred	Yes
12	Verification of users	Manual approval after talking to the operator, increases trust	Transferred	No
13	Trust score per organization	Dynamic trust score per instance	Transferred	Yes
14, 27	Multi-factor checks for disabling	Confirmation before blocking an instance key	Transferred	No
15	Minimum number of voters	Minimum number of instances	Transferred	Yes
16, 17	Visual indicators, multi-step dialogue	Unchanged within each instance	Unchanged	No
18, 19	Message signing, Transport Layer Security (TLS)	Requests signed with the instance key, TLS	Unchanged	No
20	Vote deduplication	One rule per instance and CEL expression	Transferred	Yes
21–26	Rate limiting	Rate limiting per instance	Transferred	No
28, 29	Secure authentication	Verification of the signature	Transferred	No
30	Fixed assessment values	Fixed assessment values	Unchanged	Yes
31	Deterministic tie-breaking	Deterministic tie-breaking	Unchanged	Yes
32	–	Repeat the consistency check on the server	New	Yes
33	–	Decrease the dynamic trust score of instances that lose a voting	New	Yes
34	–	Submit only the content of a rule, users are informed	New	Yes

Table 1. Re-evaluated and new mitigations for the voting with instances as voters

rule that this information is shared with the exchange server. The rules are submitted regularly by a daemon job in a request signed with the private key of the instance. The exchange server identifies the origin instance by the public key that verifies the signature and rejects requests whose signature cannot be verified or whose key is blocked.

Before submitting, the instance excludes all rules that originated from a recommendation, analogous to the voting within an instance (Section 2.2). Otherwise, a rule adopted from the exchange server would be submitted again and strengthen the majority that produced it. This filter is only applied within the instance, since the exchange server only receives the content of the rules and therefore cannot distinguish adopted rules from original ones.

For the remaining rules, the instance ensures that it sends at most one rule per CEL expression. To recognize rules with the same condition, the CEL expressions have to be compared in a normalized form. Since the instances currently store the expressions as entered and do not normalize them, the normalization is introduced as a new step before submission: each expression is brought into a canonical form, e.g. by sorting the operands of the commutative operators `&&` and `||`, so that expressions that only differ in the order of these operands become identical. Expressions that are semantically equivalent but structured differently are not recognized as identical. Rules with different conditions are not a contradiction, even if they refer to the same CVE.

- If all rules of the instance with the same CEL expression agree, the rule is submitted.
- If they contradict each other but one assessment clearly dominates, i.e. its share of the rules reaches a configurable dominance threshold, only this assessment is submitted.
- If they contradict each other and no assessment dominates, nothing is submitted.

Since the server cannot rely on the correct behavior of the instances, it normalizes the received expressions again, performs the same check and does not store contradicting rules of an instance (mitigation 32). The accepted rules are stored per instance, so that each stored rule is uniquely identified by its instance and its normalized CEL expression (mitigation 20). The CVE scope is not part of this identity, since it is derived from the expression. Each submission replaces the complete set of rules previously stored for the instance. Rules that an instance no longer submits, e.g. because they were deleted or have become contradictory, are therefore removed and no longer count as votes.

4.3 Voting on the Exchange Server

The exchange server uses the trust-weighted voting of the previous work [13], but every instance counts as one voter. The voting is not triggered by the submissions, but performed routinely by a daemon job on the exchange server. For each normalized CEL expression, the server proceeds as follows:

1. Only instances that are older than a minimum age may vote (mitigations 10 and 11).
2. Each vote is weighted by the voting weight of the instance, which combines its trust score and its dynamic trust score (mitigation 13, Section 4.4).
3. The weights are summed up per assessment. An assessment only wins if its share of the total weight exceeds the share of every other assessment by a configurable margin.
4. No rule is provided if fewer than a minimum number of instances voted or if no assessment wins by this margin, which includes ties (mitigations 15 and 31).

The winning rule is stored in a separate table on the exchange server together with a confidence value, from which the instances fetch it with their daemon job. The confidence value depends on the trust weight behind the rule: it grows with the summed voting weight of the instances that voted for the winning assessment and with the share of this weight in the total weight. A rule supported by a clear majority of trusted instances therefore receives a higher confidence value than a rule supported by many instances with little trust. If no rule wins, no rule is provided. The minimum age, the minimum number of voters and the margin are configurable. Since considerably fewer instances than organizations are expected to participate, the minimum number of voters should be chosen lower than in the previous work.

Within each instance, the retrieved rules are treated as additional rules in the local CV voting. They are evaluated against the local dependency vulnerabilities via their CEL expression like the local rules, but their vote is weighted by the confidence value instead of the trust score of an organization or project. Since they do not belong to a local organization, the organization-based mitigations of the local voting do not apply to them. The local voting thus combines the assessments of the own organizations with those of other instances, and its result is proposed to the user as a recommendation.

4.4 Dynamic Trust Score

In the previous work, trust scores were assigned explicitly by administrators [13]. In the multi-instance setting, this is not sufficient, since the administrator of the exchange server cannot assess the reliability of every instance in advance. Therefore, the trust of an instance consists of two parts:

- **Trust score:** On registration, an instance receives a low initial trust score (mitigations 7 to 9). When the administrator approves an instance, its trust score increases automatically (mitigation 12). The trust score is not changed by the voting.
- **Dynamic trust score:** The dynamic trust score reflects the voting behavior of the instance (mitigation 33). It is recalculated in every voting: starting from the maximum value, it is decreased by a configurable amount for every voting the instance loses, i.e. for every CEL expression for

which its vote differs from the winning assessment. Votings without a winning assessment, e.g. due to a tie, do not decrease it.

The voting weight of an instance is the product of both scores, and both are limited to the range $[0, 1]$. Since the dynamic trust score is recalculated instead of accumulated, it only reflects the current votes of an instance: repeated votings on unchanged rules do not penalize an instance again, and an instance whose assessment no longer loses, because it changed its assessment or the majority changed, regains its full voting weight.

The previous work already pointed out that deriving trust scores is a research problem of its own, which requires identifying suitable criteria and protecting them against manipulation [13]. This applies even more to the multi-instance setting. The approval and the voting outcome are only first parameters, chosen because they are available to the exchange server. Both have weaknesses: the approval depends on a manual assessment of the operator, and penalizing the losers of a voting reinforces the majority, so that an instance with a correct minority assessment loses voting weight as long as it remains in the minority, while a group of colluding instances that agree with each other is never penalized. Further research is therefore needed to determine which criteria reflect the reliability of an instance and how they can be protected against manipulation.

5 Implementation

The implementation consists of two parts: daemons within each DevGuard instance, which submit the rules of the instance and retrieve the voting results, and the exchange server, which stores the submitted rules and performs the voting in a daemon job, which also maintains the dynamic trust scores.

5.1 Registration and Request Signing

Each instance generates an Elliptic Curve Digital Signature Algorithm (ECDSA) key pair and is identified by a fingerprint of its public key. On registration, the instance sends its public key to the exchange server in a request signed with the corresponding private key. The exchange server verifies the signature and creates an entry for the instance with an initial status and a low initial trust score.

For signing requests, the existing mechanism of DevGuard is reused, which already authenticates requests with personal access tokens through Hypertext Transfer Protocol (HTTP) request signing [11]. Every request carries its signature and identifies the instance by its fingerprint. The exchange server looks up the public key of the instance and verifies the signature. Requests of an unknown instance, with an invalid signature or with a blocked key are rejected; all other requests are accepted, regardless of whether the instance is approved.

The status of an instance is set manually by the administrator of the exchange server. After talking to the operator, the administrator can approve the instance, which increases its trust score. In addition, the administrator can actively block the key of an instance, e.g. if the key is compromised or the instance behaves maliciously. All further requests signed with this key are rejected, and the operator has to register a new key pair (mitigations 5 and 6).

5.2 Synchronization Daemons

Within each instance, the exchange is implemented as two daemons that run routinely and independently of each other, analogous to the existing daemons of DevGuard. The first daemon collects, checks and submits the rules of the instance, the second one retrieves the voting results:

1. **Collect:** All VEX rules of the instance that did not originate from a recommendation are loaded, their CEL expressions are normalized, and the rules are grouped by their normalized expression. The normalization is a new step, since the instances currently do not normalize the expressions. The assessment is not part of the normalized form, since contradicting assessments for the same expression have to fall into the same group.
2. **Check:** For each group, the consistency check of Section 4.2 is applied with a configurable dominance threshold. Groups without a dominant assessment are skipped.
3. **Submit:** The remaining rules are sent to the exchange server in one signed request. Each rule only contains its own content (mitigation 34).
4. **Retrieve:** Independently of the submission, the second daemon fetches the current voting results with their confidence values in a signed request and stores them as remote rules.

The existing recommendation daemon of DevGuard evaluates the remote rules together with the local rules: their CEL expressions are matched against the open dependency vulnerabilities, and their votes are weighted by their confidence value. The resulting recommendations are shown to the users, who accept or reject them.

5.3 Processing on the Exchange Server

When the exchange server receives a submission, it validates the assessment of each rule against the fixed set of allowed values (mitigation 30) and applies the same normalization and consistency check as the instances again (mitigation 32), since it cannot rely on the instances to have done so. The accepted rules replace all rules previously stored for the instance in the submitted rules table, so that rules no longer submitted by the instance are removed. Since the combination of instance and normalized CEL expression is unique, an instance cannot cast a second vote for the same candidate (mitigation 20).

The voting is performed independently of the submissions by a daemon job on the exchange server, which runs routinely. In each run, the voting is performed for each normalized CEL expression as described in Section 4.3, within one transaction. Only instances whose registration is older than the minimum age are counted, each vote is weighted by the voting weight of the instance, and a result is only written if at least the minimum number of instances voted and the winning assessment exceeds every other assessment by the configured margin. The results replace the previous content of the voting results table, together with a confidence value derived from the summed voting weight of the instances that voted for the winning assessment and from the share of this weight in the total weight. Finally, the dynamic trust scores are recalculated: every instance starts from the maximum value, and the dynamic trust score of every instance is decreased by a configurable amount for each voting in this run in which its vote differs from the winning assessment, limited to a minimum of 0.

6 Evaluation

The system is evaluated qualitatively against the threat model in Section 4. An empirical evaluation, e.g. with a test suite as in the previous work [13], is left for future work.

The design addresses the main differences to the single-instance setting. Since every instance counts as one voter, large instances or instances with fabricated organizations cannot outvote smaller ones. Signed requests bind every submission to a registered key, the consistency check on the server prevents contradicting votes of an instance, and only the content of the rules leaves an instance, about which the users are informed.

Nevertheless, several risks remain:

- **Collusion:** Since every instance with a valid key may vote, an adversary can register many unapproved instances. Their low trust score limits their influence, but enough of them can still outweigh a few approved instances, and as they agree with each other, they never lose trust.
- **Majority reinforcement:** Decreasing the dynamic trust score of losing instances penalizes correct minority assessments as long as they remain in the minority and thereby strengthens the majority. Since the dynamic trust score is recalculated in every voting, this effect does not accumulate over time. Adopted recommendations are excluded only within the instance; the exchange server cannot verify that an instance applies this filter.
- **Few voters:** The number of instances is much smaller than the number of organizations, so even a low minimum number of voting instances may rarely be reached.
- **Central server:** The exchange server is a single point of failure and of trust. If it is compromised, it can distribute manipulated rules or inflated confidence values, which directly influence the local voting of every instance; this is only mitigated by users having to accept every recommendation.

The first two risks are directly related to the trust score, which confirms the need for further research on how the reliability of an instance can be determined (Section 4.4).

7 Future Work

The evaluation identified the collusion of many unapproved instances as a main remaining risk: every instance with a valid key may vote, and although a single unapproved instance has little influence, enough of them can still outweigh the approved instances. This section outlines further countermeasures against this risk, which are not part of this work.

7.1 Limiting the Influence of Unapproved Instances

The low trust score of unapproved instances only limits the influence of each single instance, so the influence of an adversary still grows with the number of instances it registers. Two extensions of the voting could prevent this:

- **Bounded group weight:** The weights of all unapproved instances could be combined and limited to a fixed share of the total weight. Registering further instances would then no longer increase the influence of this group. The bound has to refer to the weight of the approved instances, since it has no effect if only unapproved instances vote. Within the group, honest unapproved instances can still be outvoted by an adversary, so the bound protects the result of the voting, but not the vote of a single honest instance.
- **Approved vote required:** A result could only be provided if at least one approved instance voted for the winning assessment. Unapproved instances could still take part in the voting, but never produce a result on their own. However, this intensifies the cold-start problem, since no results are provided as long as only few instances are approved. Moreover, a single approved instance becomes a gatekeeper: an adversary who compromises one approved instance can then push through results together with many unapproved ones. A weaker variant only requires that at least one approved instance took part in the voting.

Both extensions complement each other: the required approved vote prevents results without any approved instance, and the bounded group weight limits the influence of unapproved instances when approved ones take part.

7.2 Costly Registration

Instead of only weighting unapproved instances lower, the registration of many instances could be made costly. An instance that is not approved manually could be required to prove the control

of a public domain, e.g. by publishing its public key under a well-known path of the domain or in a Domain Name System (DNS) record, similar to the domain validation of certificate authorities. A domain and all of its subdomains together count as one instance, so only one instance can be registered for them. Otherwise, an adversary could create any number of subdomains at no cost; this way, it has to acquire a separate domain for every additional instance.

This approach conflicts with the motivation of this work: organizations with strict confidentiality or sovereignty requirements often operate their instances internally, without public access. A well-known path would exclude these instances, whereas a DNS record only requires a public domain, not a publicly reachable instance. Furthermore, the registrable domain has to be determined according to the public suffix list rather than as the second-level domain, since otherwise all instances below a suffix would count as one. As domains are inexpensive, the deterrent effect is limited, and the proof has to be repeated regularly, since domains expire or change their owner.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[2] M. Lipow, "Number of faults per line of code," *IEEE Transactions on Software Engineering*, no. 4, pp. 437–439, 1982.

[3] A. Kurmus, A. Sorniotti, and R. Kapitza, "Attack surface reduction for commodity OS kernels: Trimmed garden plants may attract less bugs," in *Proceedings of the Fourth European Workshop on System Security*, 2011, pp. 1–6.

[4] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith, "Why do developers get password storage wrong? a qualitative usability study," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 311–328.

[5] M. Tahaei, A. Jenkins, K. Vaniea, and M. Wolters, ""I don't know too much about it": On the security mindsets of computer science students," in *International Workshop on Socio-Technical Aspects in Security and Trust*, Springer, 2019, pp. 27–46.

[6] CISA. "Minimum requirements for Vulnerability Exploitability eXchange (VEX)," Accessed: Oct. 5, 2026. [Online]. Available: https://www.cisa.gov/resources-tools/resources/minimum-requirements-vulnerability-exploitability-exchange-vex

[7] R. Ghosh, S. De, and M. Mondal, ""I wasn't sure if this is indeed a security risk": Data-driven understanding of security issue reporting in GitHub repositories of open source npm packages," *arXiv preprint arXiv:2506.07728*, 2025.

[8] DevGuard Project. "DevGuard documentation," Accessed: Oct. 5, 2026. [Online]. Available: https://docs.devguard.org/

[9] DevGuard Project. "Vulnerability mitigation strategies & VEX rules," Accessed: Oct. 5, 2026. [Online]. Available: https://docs.devguard.org/explanations/vulnerability-management/mitigation-strategies

[10] Ethereum Foundation. “Proof-of-stake (PoS),” Accessed: Oct. 5, 2026. [Online]. Available: https://ethereum.org/developers/docs/consensus-mechanisms/pos/

[11] L3montree. “DevGuard,” Accessed: Oct. 5, 2026. [Online]. Available: https://github.com/l3montree-dev/devguard

[12] L3montree. “L3montree GmbH,” Accessed: Oct. 5, 2026. [Online]. Available: https://l3montree.com/

[13] D. T. A. Quach, “Developing a trust-based knowledge-based collaborative filtering hybrid recommender system to share vulnerability management decisions in DevGuard,” Hochschule Bonn-Rhein-Sieg, Project Report, 2026.