Deriving metrics for software architectures from the "protected entry points" security patterns

Monica Buitrago, Isabelle Borne, Jeremy Buisson

Deriving metrics for software architectures from the "protected entry points" security patterns

Monica Buitrago
IRISA, Université de Bretagne Sud
Vannes, France

Isabelle Borne
IRISA, Université de Bretagne Sud
Vannes, France

Jérémy Buisson
IRISA, Académie Militaire de Saint-Cyr Coëtquidan
Guer, France

ABSTRACT

Deciding, as early as the software architecture is designed, whether the resulting system will be secure is challenging. We propose three metrics inspired by a security-related design pattern in the structural architecture model, the "protected entry points" pattern. We evaluate these metrics on the real-life Bitwarden web client and server, as well as a synthetic system.

CCS CONCEPTS

• **Software and its engineering** → **Software architectures**; *Software design engineering*; • **Security and privacy** → **Software security engineering**;

KEYWORDS

Software architecture, Security by design, Security metrics, Security patterns, Case study

ACM Reference Format:
Monica Buitrago, Isabelle Borne, and Jérémy Buisson. 2023. Deriving metrics for software architectures from the "protected entry points" security patterns. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-April 2, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3555776.3577816

1 INTRODUCTION AND RELATED WORKS

Security by design [6] is a set of practices, methodologies, tools over the whole life cycle that make a software system *intrinsically* secure. In this perspective, a catalog of security-related design patterns [3] helps architects make design decisions that prevent the subsequent introduction of vulnerabilities once the systems are developed and deployed. The "protected entry points" pattern enacts the general principles of isolation of subsystems and of privilege separation, with well-identified entry points dedicated to securing communications between subsystems, validating incoming data, enforcing access control policies, and so on. However, it is not clear whether a catalog of patterns is the right tool to effectively help designers [7].

The likelihood that software code contains vulnerabilities can be measured, using noticeably the number of parameters, of pointer arithmetic, of nested control structures, and so on [2]. The object-oriented design can be assessed by metrics such as the ratio of public attributes, the ratio of public static attributes, and metrics about mutator methods, among others [1]. The attack surface can be evaluated from the damage-effort ratio and the interactions between a system and its environment [5].

In our paper, we propose new metrics at the level of the components-and-connectors architecture, to quantify how much the architecture adopts the "protected entry points" pattern. Section 2 describes the metrics. Section 3 summarizes our experimental results. Section 4 concludes the paper with our future directions to continue our work on this topic.

2 PROPOSED METRICS

The idea underpinning the "protected entry points" pattern is that each subsystem (in the broad sense) has well-identified entry points that cannot be skirted by clients. We consider that the subsystems appear as composites in the architecture.

Let the architecture be a directed graph $\mathcal{A} = \mathcal{V}, \mathcal{E}$, where $\mathcal{V}$ is the set of components, and $\mathcal{E} \subseteq \mathcal{V}^2$ is the set of dependencies. Let C be a partition of $\mathcal{V}$ that denotes the set of composites within the architecture: we solely consider flat composites, to keep our model simple. Let $C\,(x)$, where $C : \mathcal{V} \mapsto C$, be the composite that contains the component x. Let $shortest\_path\,(a, b)$ be the shortest path from a to b, being $\perp$ if no such path exists. We note $shortest\_path^c\,(a, b)$ when the path is searched for in the subgraph c.

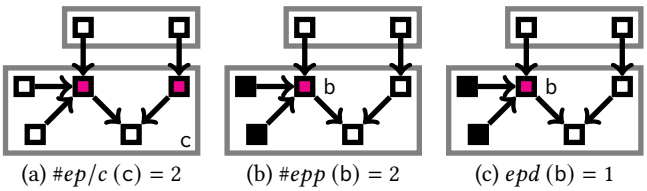
2.1 Number of entry points per composite

Let $c \in C$ be a composite, the metric counts how many components b in composite c are such that there is at least one component a that depends on b and that does not belong to composite c.

$$\#ep/c\,(c) = card\,\{b | C\,(b) = c \wedge \exists a, a, b \in \mathcal{V} \wedge C\,(a) \neq c\}$$

In the example of figure 1 (a), there are two such components that are the ones painted in red.

The higher this metric is, the more entry points let clients from the outside of c connect to the subsystem c. Each entry point must implement access control check. The more there are entry points, the more likely some of them shall lack the check, hence the system



(a) $\#ep/c\,(c) = 2$ (b) $\#epp\,(b) = 2$ (c) $epd\,(b) = 1$

Figure 1: Illustration of the metrics.

SAC '23, March 27-April 2, 2023, Tallinn, Estonia

ACM ISBN 978-1-4503-9517-5/23/03.
https://doi.org/10.1145/3555776.3577816

shall be insecure. On the contrary, when the number of entry points is low, it pinpoints that there are possibly some entry points that mix services of different privilege levels, e.g., application services mixed with administration services, a practice that is generally considered to be source of vulnerabilities, e.g., CWE-653[1].

2.2 Number of entry point predecessors

Let $c \in \mathcal{C}$ be a composite, let $b \in \mathcal{V}$ be an entry point component of c, i.e., $C(b) = c$ and $\exists a, a, b \in \mathcal{V} \wedge C(a) \neq c$, the metric counts how many components p, in the same composite c as b, are such that there is a path from p to b in the dependency graph, passing only via components that are in c.

$$\#epp(b) = card\left\{p | C(p) = C(b) \wedge shortest\_path^{C(b)}(p, b) \neq \bot\right\}$$

In the example of figure 1 (b), let b be the red-painted component. There are two such predecessors of b painted in black.

This metric characterizes how much the entry points are also internally used within the subsystem. The developer might misidentify such components as not being entry points hence not implementing access control check, resulting in deficient security.

2.3 Entry point depth in composite

This metric refines $\#epp$, using the depth of an entry point in its composite rather than the number of its predecessors. Let $c \in \mathcal{C}$, let $b \in \mathcal{V}$ be an entry point of c. The metric looks for the shortest path from any predecessor of b in c to b, then the metric returns the maximum length of these shortest paths.

$$epd(b) = max\left\{length\left(shortest\_path^{C(b)}(p, b)\right) | C(p) = C(b)\right\}$$

In figure 1 (c), let b be the red-painted component. This component b has two predecessors (in black). For each of them, the shortest path to b contains 1 edge. So, the metric returns $max\{1, 1\} = 1$.

The deeper an entry point component is in its subsystem, the more likely the developer may fail to identify it as an entry point, regardless of how many components use it directly or indirectly in the subsystem. This metric therefore ignores the breadth of the using components. In comparison to our metric $\#epp$, we expect epd to be less dependent on the size of the composite.

3 EXPERIMENT

Our metrics are at the level of the composites and at the level of their entry points. The goal of our experiments is to explore several summarizing functions to provide architecture-level quantitative measures, and to ensure that they remain stable in face of insignificant variations of the architecture.

Our experiment is based on applying the metrics to Bitwarden[2], a password vault. The figure 2 shows its coarse-grained architecture. The *server* component acts as a back-end server, that implements the services to manage the vault. The services store the vault data in a remote *database* component. The *web*, *cli*, *desktop*, *browser* and *mobile* components are the user interfaces, that provide front-ends for various kinds of devices. While not detailed in figure 2, the *server* component is decomposed into five ASP.Net Core servers, each of them being containing services based on .Net Core's dependency injection framework. Likewise, the *web* component is decomposed into services, using Angular's dependency injection framework. We do not consider the other components.

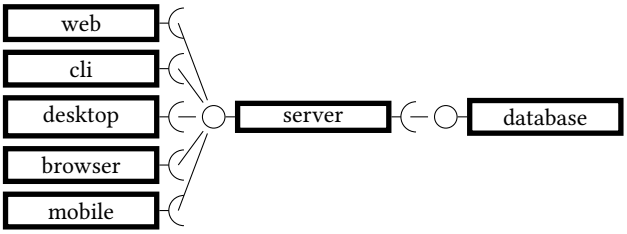


Figure 2: Bitwarden Architecture Overview.

3.1 Recover components

To begin with, we extract the classes and interfaces into a UML class diagram from the source code. From this model at the object-oriented level, the structure is abstracted as a graph of services (components) and dependencies. To do so, we detect the specific pattern of a class implementing an interface, and associated to interfaces of other class-interface pairs. Indeed, the class-interface pair is the service artifacts in the service-oriented programming models of .Net Core and Angular (used in Bitwarden).

In Bitwarden, *web* contains 775 elements (classes and interfaces) and *server* contains 1058 elements. Among them, 108 elements in *web* and 197 elements in *server* concern the components. The resulting graph, combining *web* and *server*, contains 114 components.

3.2 Recover composite components

Next, composite components are recovered thanks to a graph clustering algorithm. We pragmatically use the JGraphT[3]'s implementation of Girvan-Newman clustering algorithm [4]. This algorithm requires a parameter named K, which tells how many clusters are expected in the graph. The smallest values of K tend to result in a single cluster containing most of the components with few singleton clusters. So, these smallest values of K are irrelevant as the algorithm fails to identify the composite components. Conversely, the highest values of K tend to result in many singleton or small clusters. So, these highest values of K are irrelevant too as the resulting clusters are not composite components.

According to our experiment, the Bitwarden range of interest is: $15 < K < 50$. We also confirm that the clustering algorithm successfully recognizes *web* and *server* as composites.

3.3 Test of the metrics

Intuitively, the observed use of the "protected entry points" pattern should not depend on how fined-grained the composites are, especially in our case study since the granularity results from the clustering parameter applied to a single implementation that is reversed engineered.

When averaged, the $\#ep/c$ metric decreases when K increases (figure 3). It behaves as expected, since, at best, the number of entry points of the back-end server (the cluster with the highest

[1] https://cwe.mitre.org/data/definitions/653.html (visited on 2022-10-15).
[2] https://bitwarden.com (visited on 2022-06-27).
[3] https://jgrapht.org/ (visited on 2022-07-07).

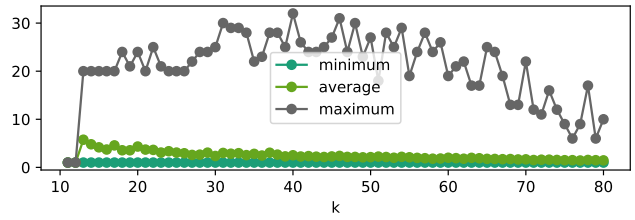


Figure 3: Analysis of $\#ep/c$ performance depending on the number of composites (Bitwarden)

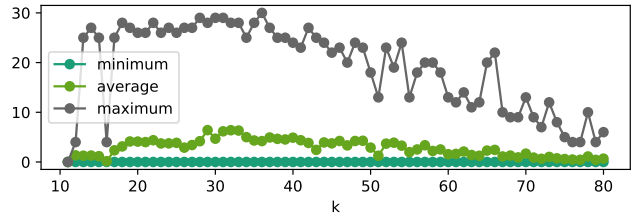


Figure 4: Analysis of $\#epp$ performance depending on the number of composites (Bitwarden)

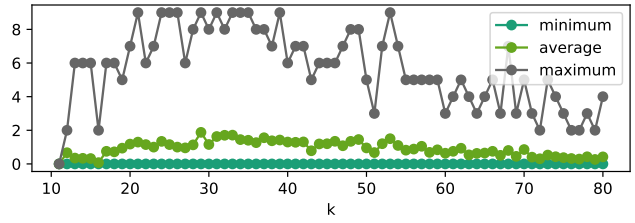


Figure 5: Analysis of epd performance depending on the number of composites (Bitwarden)

number of entry points) remains constant while additional clusters are added. This maximum is stable up to $K \leq 55$, then the value drops. We interpret this result as the back-end server being split, and therefore splitting the set of ASP.Net Core controllers over several clusters starting at $K \geq 55$ approximately. Considering only the maximum $\#ep/c$, i.e., only for the cluster with the highest value appears to be a relevant measure.

In figure 4, the $\#epp$ metric expectedly tends to decrease both in maximum and in average. Indeed, the more composites are looked for (higher value of K), the smallest the composites are, hence the fewer predecessors each entry point has. The anticipation of this fact is actually the reason for introducing the epd metric. The experiment confirms the intuition that $\#epp$ is irrelevant as it is.

In contrast with $\#epp$, in figure 5, the averaged epd metric seems independent of the value of K, up to $K = 55$, then the metric decreases as K increases. So, epd effectively improves over $\#epp$ when averaged. On the other hand, the maximum value of epd is not stable and decreases as $K \geq 40$ increases. This result appears in accordance with the observation at figure 3, stating that the clustering algorithm tends to keep all the controllers in a single cluster up to high values of K, when splitting the cluster in depth rather than the set of controllers, so does the maximum depth decreases.

4 CONCLUSION AND FUTURE DIRECTIONS

This paper reports our study towards metrics to measure the security level at an architecture model. We propose three metrics targeted at the entry points of the composite components, rooted in the "protected entry points" pattern [3]. The first metric $\#ep/c$ measures the number of entry points per composite: the higher it is, the more components the security concerns are spread over; but, a too low value for $\#ep/c$ is symptomatic of improper compartmentalization or of mixing privilege levels. The second metric $\#epp$ measures the number of dependents per entry point: the higher it is, the deeper the entry point is nested in its composite and the higher the risk that the entry points are not identified as such. The third metric epd refines $\#epp$ by considering the depth rather than the number of dependents.

In this preliminary study using a real-life application, we test whether the metrics are stable regardless of the decomposition of the architecture into composite components. Indeed, we expect the security level not to depend on how much the architecture is decomposed, especially in our case study where the composites are recovered by clustering from a reverse-engineered implementation. The max $\#ep/c$ and averaged epd metrics give promising results, while the $\#epp$ metric does not seem relevant as it is. Still, epd appears to fix the issue of $\#epp$.

In our future work, we intend to further study the normalization of these metrics. Next, we plan an experiment to more specifically relate the metrics with security. To do so, we intend to check the correlation between the metrics value and the security level, assessed by some security experts. This study will require additional software architectures to enable the comparisons.

REFERENCES

[1] Bandar Alshammari, Colin Fidge, and Diane Corney. 2010. Security Metrics for Object-Oriented Designs. In *2010 21st Australian Software Engineering Conference*. 55–64. https://doi.org/10.1109/ASWEC.2010.34

[2] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 60–71. https://doi.org/10.1109/ICSE.2019.00024

[3] Eduardo Fernandez-Buglioni. 2013. *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*. Wiley.

[4] M. Girvan and M. E. J. Newman. 2002. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences* 99, 12 (June 2002), 7821–7826. https://doi.org/10.1073/pnas.122653799

[5] Pratyusa K. Manadhata and Jeannette M. Wing. 2011. An Attack Surface Metric. *IEEE Transactions on Software Engineering* 37, 3 (May 2011), 371–386. https://doi.org/10.1109/TSE.2010.60

[6] Michael Waidner, Michael Backes, and Jörn Müller-Quade. 2014. *Development of Secure Software with Security By Design*. Technical Report SIT-TR-2014-03. Fraunhofer Institute for Secure Information Technology.

[7] Koen Yskout, Riccardo Scandariato, and Wouter Joosen. 2015. Do security patterns really help designers?. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. 292–302. https://doi.org/10.1109/ICSE.2015.49