# Dinis Cruz's Research on API Security (2009-2025)

*by Dinis Cruz and ChatGTP Deep Research, 2025/09/01*

## Introduction

API security has been a persistent theme in Dinis Cruz's work, spanning early insights in 2009--2010 through to innovative ideas in 2025. His contributions center on how to **test and analyze web services/APIs securely**, often by blending static analysis, dynamic testing, and developer-centric tooling. A recurring motif is that *every API and web service is inherently vulnerable at some level* -- thus requiring proactive design, rigorous testing, and automation to mitigate risks[1][2].

Cruz's writings emphasize creating security tests early, making security "invisible" to developers (through automation), and leveraging advanced techniques (like AST manipulation and knowledge graphs) to understand and secure APIs. This debrief summarizes Cruz's key publications on API security (with publication dates) and distills their core ideas, providing AppSec professionals and next-gen AI-driven API security teams a cohesive view of his work. *(Authored by Dinis Cruz and ChatGPT Deep Research.)*

## Key Publications and Concepts on API Security

The table below lists Cruz's main writings on API security and testing, with a brief description and their key ideas:

| Publication (Date) | Summary & Key Ideas |
|---|---|
| **"Every API is (at some level) vulnerable" (Blog – Jan 9, 2010)** | Introduces the premise that any API/function with capability likely harbors vulnerabilities. Advocates mapping out "layers" of vulnerable functions and tracing if/when they become externally accessible (i.e. exploitable). Stresses that secure API design should *wrap internal flaws* so they can't be abused externally (e.g. .NET's Code Access Security treating partial-trust boundaries as the attack surface). Published 2010-01-09. |
| **"ESTAPI idea" (Blog – Jun 2011)** | Proposes the **ESTAPI** (Enterprise Security Testing API) concept. Envisions a standardized way to expose or automate security tests for web apps/services. *Key idea:* make security testing a first-class function via an API, so that dynamic tests and security checks can be uniformly applied. *(Published 2011-06; part of "ESTAPI" series of 5 posts.)* |
| **"Fixing/Encoding .NET code in real time (Response.Write)" (O2 Blog – Nov 7, 2011)** | Demonstrates runtime patching of vulnerable code using OWASP O2 Platform. Hooks the `.NET Response.Write` to auto-encode output, preventing XSS on the fly. Emphasizes making security **"invisible/automatic" for developers** by fixing issues in dev environments transparently. Key idea: security frameworks/tools should actively prevent vulns (and allow testing such fixes) during development. |

| Publication (Date) | Summary & Key Ideas |
| --- | --- |
| **"If you're not blowing up the database, you're not testing" (Blog – Apr 2012)** | Argues that effective web service testing must be aggressive. If tests never break anything (e.g. never cause DB errors or crashes), they are too shallow. Recommends testers simulate malicious and extreme conditions – even at risk of "blowing up" components – to reveal security weaknesses. *(Published 2012-04; underscores the need for thorough, adversarial testing mindset.)* |
| **"Journey into testing WebServices in TeamMentor" (Blog – Apr 2012)** | A narrative of initial attempts at testing the **TeamMentor** product's web services. Documents setting up a testing environment and early findings. Likely covers using Python scripts and SOAP/REST clients to enumerate API calls. *Key ideas:* the importance of understanding an API's functionality before testing and dealing with real-world complexities (authentication, state, etc.) during security testing. *(Published 2012-04.)* |
| **"First, you create tests for WebServices..." (Blog – Apr 2012)** | Advocates a *test-first approach* for web service security: before or as you build a web API, create a suite of security tests. By writing tests that target each endpoint (including negative cases), developers can catch vulnerabilities early. Emphasizes that security requirements should be codified as tests from the start – similar to Test-Driven Development but for security. *(Published 2012-04.)* |
| **"Roadmap for testing WebServices" (Blog – May 2012)** | Outlines a structured plan for web service testing. Proposes steps such as: **1)** Enumerate and understand all API endpoints; **2)** Write unit and integration tests for each (covering auth, input validation, business logic); **3)** Use automation (scripts or tools) to fuzz inputs and simulate abuse; **4)** Incorporate tests into CI. Likely highlights needed tooling and roles (QA, AppSec) for each step. *(Published 2012-05.)* |
| **"What is the formula for WebServices security testing?" (Blog – May 2012)** | Explores whether a "formula" or repeatable methodology can be applied to test any web service. Suggests combining **static analysis** (to find hidden endpoints or dangerous calls) with **dynamic testing** (to actually try attacks). The "formula" involves mapping trust boundaries, identifying inputs/outputs of each service, then systematically verifying authentication, authorization, input handling, and error handling for each. *(Published 2012-05.)* |
| **"Big security challenges with creating an API" (Blog – Jun 2012)** | Reflects on pitfalls in designing secure APIs. Topics include: managing authentication/authorization complexity, avoiding data over-exposure, versioning and backward compatibility introducing security debt, and the difficulty of providing flexibility to developers without opening attack surface. Uses examples (likely from building TeamMentor's API) to illustrate how design decisions can inadvertently create vulns. *(Published 2012-06.)* |
| **TeamMentor Web Services Testing Series (A. Doraiswamy & Cruz – Apr 2012)** | A four-part blog series (April 28, 2012) by Arvind Doraiswamy with Dinis Cruz's help, detailing the process of testing **TeamMentor**'s SOAP/REST web services. It covers setting up a Python test harness (using *Suds* for SOAP), writing scripts for each API method, and logging/visualizing results. The series uncovered auth issues and logic flaws, and produced an **authorization mapping spreadsheet** for TeamMentor's API. Key lesson: a systematic, script-driven approach can reveal subtle authorization gaps and drive the creation of proper access-control matrices. |

| Publication (Date) | Summary & Key Ideas |
| --- | --- |
| **"Creating an API to create WebServices (on the fly)" (Blog – May 2013)** | Describes an innovative **meta-API** that can dynamically generate new web service endpoints. Cruz explored how an application could accept high-level definitions and spawn actual web APIs at runtime. The experiment (circa May 2013) demonstrated both the power and security implications of on-the-fly API generation. *Key ideas:* automation in API creation could help testing (by generating test endpoints or mocks), but it must be carefully secured to avoid abuse (since an API that creates APIs could be misused if not locked down). |
| **"O2 Platform's MethodStreams (2010): Open Source SAST Engine" (Tech Blog – Feb 11, 2025)** | A retrospective on the **MethodStreams** feature of OWASP O2 Platform (originally developed 2010). MethodStreams automatically **traverses a web service's call tree** to aggregate all its relevant code into a single view. By extracting the AST of each method in the execution path, it produces a consolidated file with only the code that implements that API endpoint. This greatly streamlines code review and security analysis for large apps: e.g., a web method that spanned 20+ files and 50k lines could be reduced to a 3k-line snippet of just what matters. The document (2025) highlights how MethodStreams exemplifies using static analysis to aid **targeted API review**, and it encourages modern tools to adopt similar programmatic AST manipulation for security. |
| **"Semantic Knowledge Graphs for LLM-driven Source Code Analysis" (Article – May 29, 2025)** | Explores the use of **semantic knowledge graphs** in combination with Large Language Models to analyze source code (including APIs). Cruz notes that capturing code facts (functions, data flows, auth checks, etc.) in a graph can help an LLM reason about security properties of an API. This approach could automate detection of missing authorization or data exposure by having the LLM traverse a knowledge graph of the application's API structure. *(Published 2025-05-29; indicates the forward-looking integration of AI/LLM into API security testing.)* |
| **"AST (Abstract Syntax Tree)" (Medium blog – Jul 2017)** | Discusses the power of ASTs for code understanding and security. Shares examples where AST parsing enabled writing tests to ensure every exposed web service method calls an authorization routine. Introduces **code transformations** like MethodStreams and automated refactoring to fix vulnerabilities. Emphasizes that developers can leverage AST tooling to enforce security requirements (e.g., "no API method without input validation" can be checked via AST-based unit tests). This ties back to his earlier work by showing how *automation and code parsing* can systematically improve API security. |
| **SecDevOps Risk Workflow (Leanpub e-book – v0.1 draft, 2017)** | An early-stage book where Cruz frames how to embed security into DevOps pipelines. While not solely about APIs, it reinforces relevant ideas: *security unit tests as part of CI*, "security champions" in dev teams, and risk acceptance workflows. One notable point: encouraging high code coverage in tests so that one can see exactly what parts of an app (or API) are exercised by tests versus which are "dark" (unhit by any test). In the API context, this means ensuring no endpoint is left untested or with dead code that could become a vulnerability. Cruz also highlights making security decisions visible and automating as much as possible – principles that next-gen API security tools should incorporate. |

*(Table: Dinis Cruz's publications on API security, with dates and key ideas.)*

# Evolution of Themes in Cruz's API Security Research

## Early Insights: "Every API is Vulnerable" (2010–2011)

Cruz's early writings set the tone by asserting that **any API function can be a weakness** if it processes untrusted input or performs sensitive actions. In *"Every API is (at some level) vulnerable"* (Jan 2010), he notes that when doing code reviews, we often debate whether a given API is *inherently* vulnerable -- and his conclusion is usually "yes, if it does something powerful"[1]. For example, a data-layer method that executes an SQL query from a string is *"vulnerable by design to SQL injection"* -- the real question is whether an attacker can reach it[15]. This led to the practice of **mapping internal APIs and tracing call chains** to see if untrusted users can invoke them. Cruz recommended identifying layers of vulnerabilities and *"mapping them upwards"* through consuming functions until you either hit an entry point or determine the issue isn't exploitable[4]. This approach laid a foundation for later ideas like MethodStreams (which automates gathering those call chains).

In 2011, Cruz proposed the **ESTAPI (Enterprise Security Testing API)** concept. At a high level, ESTAPI was about providing a uniform interface or service in applications for security testing. While details were still exploratory, the goal was to make it easier to *invoke security checks or inject test scenarios* via an API. This reflects Cruz's philosophy of baking testing capabilities into systems, so that testers (or even malicious hackers) don't have to "go around" the application to test it -- instead, the system would openly present a testing surface. Although ESTAPI did not become a standard, the idea prefigured today's trends of self-test endpoints or integrated scanning hooks in web apps.

Also around this time, Cruz was deeply involved in OWASP's O2 Platform tool. One notable blog from Nov 2011 describes using O2 to **patch a live application against an XSS vulnerability** by auto-encoding outputs. In *"Fixing/Encoding .NET code in real time"* (2011) he shows a proof-of-concept where, in a development environment, a dangerous call ( `Response.Write` of unencoded data) is intercepted and fixed on the fly. He argues this kind of automation could make security **"invisible" to developers by happening automatically**[5]. The broader implication for API security is that frameworks and platforms should help developers by transparently handling common security tasks (like output encoding or input validation), thereby reducing the chance that an API is deployed with a trivial vulnerability. This early focus on automation and developer experience carries into his later SecDevOps work.

## 2012: Developing a Methodology for Testing Web Services

In 2012, as web APIs and SOAP/REST services were becoming core to applications, Cruz's blog output on the topic peaked. He published a series of posts that effectively build a **methodology for API security testing**. A key principle was: *start by writing thorough tests*. In *"First, you create tests for WebServices"* (Apr 2012), he suggests that before trying to secure or break an API, one must have a baseline of **functional tests** that cover each endpoint and use-case. This ensures the tester understands expected behavior and can detect when a security test causes a deviation or failure. It's an approach analogous to test-driven development -- here used for security, to the point of recommending that if you're developing a web service, write its security tests alongside its features.

Cruz's *"Journey into testing WebServices in TeamMentor"* and related 2012 posts put this into practice. TeamMentor was an application with its own web service API, and along with colleague **Arvind Doraiswamy**, Cruz helped create a Python-based test harness to exercise it. They leveraged SOAP clients (the **Suds** library in Python) to enumerate all web service methods, call them with various inputs, and check responses. Through this journey, they produced artifacts like a spreadsheet mapping each web service method to the user roles allowed to invoke it[6] -- essentially an **authorization matrix**. This was in response to real issues: for example, they discovered cases where certain API calls lacked proper authorization checks, which such a matrix would quickly highlight. This work underscored the value of systematically *mapping "who can do what" in an API* and

using that as both a guide for testing and a deliverable to the dev team. (In fact, Cruz posted on StackExchange about creating these authorization mapping spreadsheets[16] to encourage others to do the same for their APIs.)

Another notable mantra from 2012 was encapsulated in the provocatively titled *"If you're not blowing up the database, you're not testing"*. Here, Cruz stresses the importance of **aggressive test scenarios**. In practice, when testing a web service, this means doing things like sending extremely large payloads, deliberately malformed data, or bizarre sequences of requests -- essentially trying to make the API or its backend components fail. If none of your test cases ever trigger a crash, heavy load, or an error in the logs (e.g., *blowing up the database* with a crazy query), then your testing may be too superficial. While this approach can cause downtime in a production environment (hence should be confined to dev/test setups), the philosophy is that only by pushing systems to their limits do you expose security-critical weaknesses (such as SQL injection flaws that could dump an entire database, or logic bugs that corrupt data). Cruz's point was that **robust APIs should be able to handle abusive or unexpected inputs gracefully** -- and if they can't, it's better for a tester to discover that than an attacker.

By May 2012, in *"What is the formula for WebServices testing?"* and the *"Roadmap for testing WebServices"*, Cruz attempted to generalize these lessons. He broke down API security testing into repeatable steps -- from **reconnaissance** (understand the API surface: endpoints, parameters, technologies) to **test design** (cover authentication, authZ, input validation, business logic, error handling) and **execution** (automation tools, fuzzers, etc.). An interesting aspect he highlighted was combining **static analysis with dynamic testing**. For instance, static analysis might find a hidden API endpoint or developer-backdoor function not documented; testers could then include that in their dynamic test plan. Or static analysis could reveal that a web method calls a database query with no parameterization -- flagging it as a SQL injection candidate to verify via dynamic test. This blended approach was ahead of its time, foreshadowing the modern IAST (Interactive Application Security Testing) tools that instrument apps to get the best of both worlds. Cruz's writing from this period is essentially a call to be *thorough and systematic*: use every tool available (code review, unit tests, integration tests, fuzzing, runtime monitoring) to poke at your APIs from all angles.

In *"Big security challenges with creating an API"* (Jun 2012), Cruz took a step back to the design perspective. Here he enumerated common hard problems when developing APIs securely. For example, **authentication & authorization**: providing a smooth developer experience (DX) often conflicts with enforcing strict security. If an API is too permissive by default, it's a risk; if it's too locked-down, developers might hack around it. **Excessive data exposure** was another challenge -- many APIs tend to return more data than necessary (e.g., returning full user records where only names are needed), which can accidentally leak sensitive info. Cruz also discussed **state management** and how stateful vs stateless designs carry different security considerations (like replay attacks or race conditions in state changes). The takeaway was that designing an API is not just an engineering problem but a security one: choices made early (URL structure, payload format, use of HTTP verbs, etc.) all can introduce or mitigate classes of vulnerabilities. This blog served as a checklist for API designers to think like an attacker *from the design phase onward*.

## Advanced Tooling: MethodStreams and Dynamic API Generation (2013)

By 2013, Dinis Cruz's focus expanded to more advanced ideas, leveraging his tooling background. In *"Creating an API to create WebServices"* (May 2013), he explored a meta-level concept: What if an application exposed an API that let you spin up *new API endpoints* on demand? This sounds esoteric, but it had practical motivation. For testing purposes, such a capability could let one create test endpoints (perhaps to echo inputs or perform specific tasks) without redeploying the server -- useful for hooking into an application's internal functions dynamically. It also mirrored how cloud services and integration platforms were evolving -- programmatically defining endpoints and their logic. Cruz's experiment likely used scripting (maybe an O2 Platform script or reflection in .NET) to register new web service methods at runtime. The **security angle** is double-edged: on one

hand, this could enable *on-the-fly security tests* or probes. On the other, if such a feature were present in production (even inadvertently, like a debug interface), an attacker could abuse it to create backdoor methods. Cruz acknowledged this risk -- it's a reminder that meta-programming features must be guarded. The broader point from this exploration was visionary: *APIs can be made extensible and dynamic, but doing so safely requires careful design*. This anticipated today's serverless and plugin-based architectures, where endpoints and functions come and go dynamically.

Another significant development was the deeper discussion of **MethodStreams** in the context of the O2 Platform. Although MethodStreams had been built in 2010, Cruz documented it thoroughly in a February 2025 post, reflecting on its impact. The MethodStreams approach is worth reiterating: given a particular entry-point method (say a REST API handler or SOAP operation), O2's MethodStreams module will find *all* methods it calls, then all methods those call, and so on -- essentially the full call graph -- and then **merge all those code fragments into one linear stream** of code[8][17]. The result is a single file or view showing exactly what that API does from start to finish, including data validations, business logic, and data access. In a large enterprise application, where understanding a single web request might otherwise demand opening dozens of files and classes, MethodStreams hugely simplifies the reviewer's task[10]. For security testing, this means you can more easily spot where a validation is missing or where an unsafe call is made, since the context is preserved in one place. Cruz pointed out that this capability *"made a massive difference when doing code reviews"*[18] -- it was like having an X-ray of the application's internals focused on one API endpoint. In today's terms, this aligns with **trace-based static analysis** or **code property graphs** that security researchers use; Cruz was implementing a form of that a decade earlier. His advocacy in 2025 is for toolmakers to revive and modernize this idea: use ASTs and parsing to help engineers and AI systems visually and programmatically understand the full scope of an API's behavior. It's an approach that can complement traditional testing by revealing hidden paths (e.g., a method might indirectly call a dangerous API deeper in the stack -- MethodStreams would surface that, guiding the tester where to focus).

## Recent Work: AI, Knowledge Graphs, and Integrating Security into DevOps (2017--2025)

In the latter part of the decade, Cruz's research pivoted to how emerging technologies like machine learning and knowledge graphs could tackle longstanding security problems -- including API security. His 2017 Medium article on *"AST (Abstract Syntax Tree)"* not only recaps the MethodStreams and AST-driven testing concepts, but also demonstrates writing **automated tests from ASTs**. For example, he posits an organization might require that *"every exposed web service method must call an authorization check."* Enforcing this manually is hard; but Cruz shows you can parse the code, find all public web methods, and then programmatically verify that within their AST there is a call to the `Authorize()` function (for instance)[11][12]. He even suggests generating documentation from these AST-based tests -- turning them into living standards. This approach empowers the next generation of **"secure by design"**: if developers can easily create such AST-driven unit tests, they get immediate feedback when they violate an architectural security rule (much like a unit test failing). It's an evolution of the "test-first" idea, augmented with modern parsing and automation.

Moving into 2025, Cruz wrote about *"Semantic Knowledge Graphs for LLM-driven Source Code Analysis."* This cutting-edge piece suggests that to harness AI (Large Language Models) for code security, we should feed them richer, structured information -- namely **knowledge graphs** that encode facts about the code. In context of API security, imagine a graph where nodes are API endpoints, data models, permissions, internal functions, etc., with edges representing calls, data flows, or access requirements. An LLM with this graph could answer complex questions (e.g., "Which API endpoints can modify customer records and do they all require admin tokens?") without having to read raw code line by line. This resonates with Cruz's long-standing vision of *making security knowledge consumable by tools at the exact moment needed*[19][20]. Back in 2010, he lamented that security advice buried in documentation is "almost as good as non-existent" unless it's available in context[19] -- fast forward to 2025, and he's effectively proposing to codify that knowledge in graphs that AI agents can use

during code analysis. It's a direct bridge from the static/dynamic analysis integration he championed into the AI era: the static facts (graph) combined with dynamic reasoning (LLM) might finally crack some hard problems of scale in API security analysis.

Finally, Cruz's work on **SecDevOps** (Secure DevOps) provides a philosophical backdrop. He consistently argued that security testing (including for APIs) must be integrated into the development lifecycle -- not a one-off external audit. In his SecDevOps Risk Workflow book, he highlights practices like: treating security findings as normal bugs, using CI pipelines to run security tests, and giving developers self-service security tools. For API security, one practical implication is to include API security scans in CI (e.g., automated fuzzing or dependency checks every build). Another is to measure test coverage for API endpoints -- ensuring that any new API functionality is accompanied by corresponding security tests. Cruz mentions the idea of tracking how much of an application's code is exercised by tests, and warns against leaving large swaths untested especially in APIs exposed to the outside[21][22]. This quantification of coverage can drive risk discussions: a CISO could ask, "We have 100 APIs, but only 70% of them are covered by automated tests -- what about the remaining 30%?" Making such gaps visible is key to *managing API risk continuously*, rather than in the spurts of occasional pen-tests.

In summary, across 15+ years, Dinis Cruz's contributions to API security revolve around **proactive testing, developer enablement, and smart automation**. From manual code review tips in 2010, to building custom tools in 2012, to leveraging ASTs and AI by 2025, his work consistently pushes towards making it easier to understand and secure the complex web of functionalities that modern APIs expose.

## Takeaways for Next-Generation API Security Solutions

Cruz's research offers a wealth of guidance for those building the next generation of API security products -- including GenAI-driven platforms and traditional AppSec tools. Here are key takeaways and how they align with what CISOs, AppSec teams, and developers are seeking in 2025:

- **1. Security Testing Must Be Automated and Integrated** -- One of Cruz's core messages is to embed security tests into the development process. Tools should make it trivial to create and run API security tests as part of CI/CD. For example, an ideal product might automatically generate baseline security test cases for each new API endpoint (much like Cruz manually wrote tests for each web service in 2012). This addresses CISO concerns by providing continuous assurance, and developers appreciate it when security checks run seamlessly in their pipeline rather than as last-minute firefights.
- **2. *"Think how someone could abuse it"* -- threat modeling at design time** -- Cruz often starts with the mindset of an attacker or an abuse-case (famously noting: *"when you develop your awesome, flexible, dynamic thingamajiggy, think about how someone could abuse it"*[23]). Next-gen solutions should help teams model threats to their APIs early. This could mean providing templates for common API threat scenarios (like broken object access, injection, auth bypass) and integrating those into user stories or design reviews. A product that can take an API definition (e.g. OpenAPI spec) and automatically highlight risky elements (e.g. "This endpoint exposes user email addresses -- is that intended?") would embody this principle. CISOs value this because it reduces the chance of design-level flaws; developers value not having to manually brainstorm every possible abuse case thanks to assistive tools.
- **3.** Every **API is vulnerable -- discover all and monitor them** -- An implication of "every API is vulnerable" is that organizations need visibility into all their APIs and an understanding of their exposure. Tools should emphasize API discovery (including shadow or undocumented APIs) and mapping of internal dependencies. Cruz's approach of mapping internal function call graphs and building authorization matrices can inspire features: for instance, an API security platform that

automatically builds a **knowledge graph of the application's API endpoints, the data they touch, and the guards (auth, validation) around them**. This directly appeals to CISOs' needs for inventory and risk assessment, and helps AppSec teams pinpoint weak links. Modern "API security posture management" solutions echo this, and Cruz's work reinforces how crucial comprehensive mapping is.

- **4. Combine Static and Dynamic Analysis for API code** -- As shown by MethodStreams and AST-based testing, static analysis can uncover deep issues and guide dynamic tests. Next-gen products should blend these techniques. For example, if static analysis finds that an API method does not validate an input, the tool could then prompt a dynamic fuzz test on that parameter to confirm exploitability. This two-pronged approach yields more reliable results (fewer false positives, more confirmed vulnerabilities) -- something CISOs want to see in reports. Developers benefit too, as they get concrete evidence of issues (a failing test or proof-of-concept) rather than theoretical warnings. GenAI can assist here by interpreting static analysis findings and suggesting or even executing relevant dynamic tests, much like an expert pen-tester would -- a concept very much in line with Cruz's vision of augmented analysis.

- **5. Developer-Friendly Remediation and "Invisible" Security** -- A consistent theme in Cruz's writing is making life easier for developers to do the right thing. Whether it's auto-encoding outputs or providing ready-made test scripts, the idea is to reduce friction. Products targeting API security should therefore focus not just on finding problems, but also on *fixing* them or preventing them. This might include secure frameworks that automatically handle common vulnerabilities (for instance, a library that sanitizes inputs to DB queries to prevent SQL injection, much as Cruz's runtime fix did for XSS). Another example is providing **code-fix suggestions** when a vulnerability is found -- e.g., "This API endpoint is missing an authorization check; here is a patch to add one." Development teams are more likely to embrace security tools that save them time and integrate with their workflow (IDE plugins, pull request scanners, etc.). By striving for that "invisible/automatic" feel -- where security is embedded and doesn't slow down development -- vendors will meet the approval of both dev teams and security officers.

- **6. Leverage AI and Knowledge Graphs for Scale** -- Modern applications might have hundreds of microservice APIs, which is challenging to analyze manually. Cruz's recent research into knowledge graphs and LLMs provides a blueprint for tackling this complexity. Next-gen solutions should utilize AI to correlate information across APIs and detect patterns humans might miss. For example, an AI-driven analysis might identify that *two different APIs together could be abused in a logic attack* (one API exposes a user's ID and another, separate API accepts that ID to delete an account -- together they create a vulnerability). A knowledge graph that links data flows and privileges can feed an AI to surface such issues. Importantly, AI should also help with **prioritization** -- something CISOs care deeply about. Among thousands of API endpoints, which pose the highest risk? Cruz's mindset of evidence-based risk (e.g., focusing on APIs that connect to critical data or have known vulnerability patterns) can be operationalized by AI analyzing the code and usage telemetry. The goal for new products should be to reduce noise and highlight *meaningful, context-rich security findings*, effectively applying the kind of deep insight that Cruz demonstrated manually, but at machine speed and cloud scale.

- **7. Continuous Verification and Observability** -- Finally, Cruz's approach implies that security isn't a one-time event. Just as he advocates continuous testing and even runtime checks, new API security companies should think in terms of *ongoing verification*. This could mean runtime protection (RASP for APIs) that monitors for abnormal behavior and even blocks attacks. It also means providing dashboards where AppSec teams and product owners can see the "security health" of their APIs in real time -- test coverage, known vulnerabilities, recent attack attempts, etc. By continuously measuring and reporting (e.g., "95% of our APIs have up-to-date security tests and no critical vulns"), security becomes a quantifiable aspect of quality, which is exactly how Cruz frames it in SecDevOps

discussions[24][25]. Such capabilities align with what management (CISOs) want -- assurance over time -- and what developers need -- feedback loops to improve.

In conclusion, Dinis Cruz's body of work on API security serves as a road map for building effective solutions in this space. The emphasis on **proactive testing, automation, developer enablement, and intelligent analysis** is now reflected in the market's expectations for API security tools. By learning from these contributions, next-generation products can better empower developers to create secure APIs from the start, give AppSec teams the deep visibility and automation they need, and ultimately provide CISOs the confidence that their organization's APIs are not the weakest link. The future of API security, as Cruz's research suggests, will belong to those who integrate security seamlessly into the DNA of software development and leverage smart technology to stay ahead of attackers.

**Sources:** Dinis Cruz's personal blog posts and articles[1][7][23][6][5], as summarized and interpreted above.


[1] [2] [3] [4] [15] Dinis Cruz Blog: January 2010

https://diniscruz.blogspot.com/2010/01/

[5] Fixing Code « OWASP O2 Platform Blog

https://o2platform.wordpress.com/category/fixing-code/

[6] [16] web service - Is there a spreadsheet/template for Mapping WebServices Authorization Rules? - Information Security Stack Exchange

https://security.stackexchange.com/questions/14598/is-there-a-spreadsheet-template-for-mapping-webservices-authorization-rules

[7] [8] [9] [10] [11] [12] [13] [17] [18] AST (Abstract Syntax Tree). AST (Abstract Syntax Tree) is a graph... | by Dinis Cruz | Medium

https://medium.com/@dinis.cruz/ast-abstract-syntax-tree-538aa146c53b

[14] [21] [22] [24] [25] Read SecDevOps Risk Workflow | Leanpub

https://leanpub.com/secdevops/read

[19] [20] Dinis Cruz Blog: Can we please stop saying that XSS is boring and easy to fix!

http://diniscruz.blogspot.com/2010/09/can-we-please-stop-saying-that-xss-is.html

[23] docs.huihoo.com

https://docs.huihoo.com/javaone/2014/CON2488-RESTing-on-Your-Laurels-Will-Get-You-Pwned.pdf