DevSecOps 1010

How MTSI is Investing in the New USAF DevSecOps Initiative

Prepared by

MTSI
MODERN TECHNOLOGY SOLUTIONS, INC.®

Chris Wilson
Ben Winks

mtsi-va.com

Table of Contents

Introduction

The United States Department of Defense (DoD) pivoted its traditional software development practices to take advantage of the best practices of the commercial world. For years, words like “Agile” and “DevOps” have been hastily crammed into proposals alongside a traditional waterfall schedule of development and delivery. This juxtaposition between lip service to industry best practices and “status quo” schedules has hindered real progress - until now. Enter Nicolas Chaillan, a champion of modernizing the technologies, policies, and practices to develop and deliver software to the DoD. With the DoD changing course, the game has fundamentally changed. It is time to get on board, play ball, or stay out of the way during this rapid shift away from the slow moving, expensive methodologies of the past. From an engineer’s perspective, this is a wonderful thing. It allows those who have already been adhering to these standards to look forward to a bright future - a future where the tried and true practices being used no longer have to be obfuscated to meet the specific, old-school criteria that a proposal dictates.

The purpose of this whitepaper is twofold; on one hand, there seems to be a real air of mystery surrounding the term DevOps, let alone Dev*Sec*Ops. A technical deep dive would really only cover the technologies that enable the people and processes, while an analysis of the processes and concept of DevOps as a philosophy would not get us close enough to paint a clear picture of how to get there. Our hope is that by the end of this paper, readers from a disparate array of backgrounds - from business development and executives to program managers and engineers - should have a much more intuitive understanding as to what DevOps really is at its core. The style of this paper is relatively informal, but to quote a sentiment often attributed to Albert Einstein, “If you can’t explain something simply, you don’t understand it well enough”.

Articulate speech is often confused with profundity (ironic, eh?), so we will use real world examples, analogies, and simplified concepts to hopefully give readers something to hold onto in terms that they may be more familiar with.

DevOps

What it is

So, what is DevOps? Well, ask 5 engineers and you will get 5 answers and an argument to boot. In essence, the word itself is an excellent example of what it is trying to achieve; a portmanteau of "Development" and "Operations", mashing them together in much the same way as the practices it espouses would mash together those traditionally separate teams. Development teams develop software, and Operations teams deploy it and make sure it stays up and running smoothly. The key idea here - and we will touch on the 'how' later - is to get these two teams in sync with one another. Anyone involved in software development or operations has seen the dumpster fires and long nights that result from a breakdown in communication between the Development and Operations teams. A bug in production? Operations points at Development, after all, they wrote the code! The Development team asks loudly why it wasn't caught in testing and deployment, that's the whole point of an Operations team! And unfortunately, they are both semi-correct, however unproductive the argument may be.

A solution! Why not combine the two teams, and that will fix it all, right? Well…not really. It will certainly help communication, but it's still a case of "Team A make a widget, give it to Team B who will set it up." There will inevitably be some instability and crunch time as patches and hotfixes are rushed out to avert the crisis of the week. Before we can really make an effective push towards this new paradigm, we must look at what we are trying to accomplish and reimagine how we go about achieving that end.

The days of a monolithic software service sitting on a massive server held together by a crack team of engineers, fueled by their stimulant of choice and years of intricate, specific knowledge of that exact setup...is drawing to a close. As our hardware has become further abstracted from us with virtualization and containerization, software services have begun to shrink from the paradigm of a “throw everything into a single code base until it inevitably collapses under its own weight” to that of a much more decoupled “write a dedicated service for every domain of functionality in the system”. Instead of a massive codebase that serves a webpage, handles database connections, manages users, and hosts a remotely accessible API, we break these clearly defined domain boundaries into a service for each. This has many benefits - each individual service can be written in a language that suits that task, regardless of how the other services are implemented. It shrinks the codebase into a more manageable size, reduces complexity, and allows the update of a single service to occur without impacting the rest of the system.

Breaking things apart does not necessarily guarantee that each domain will reduce its internal complexity; in this context, micro is not synonymous with small. But when you reduce the scope of a service to only a single domain, its complexity is isolated to itself. only internal. By decoupling large, focused pieces of functionality into separate components, intercommunication shifts to the boundaries with an interface. As noted in the next section, this allows us to do something quite exceptional when it comes to deploying our whole solution as a collection of independent components.

Enter the concept of Continuous Integration and Continuous Delivery (CI/CD). These smaller components are much better when it comes to testing, maintaining, and deploying the codebase. Of course, it is! There is less complexity per piece to manage. But we simply kicked the complexity down the road to the Operations team. Now they must handle multiple deployments and troubleshoot the inevitable breakdown of communication between two vital pieces. If you have a mature setup, that additional complexity increases per environment that you have. What if we could shift this back to the middle? What if we could deploy to production with a twofold confidence: that the software we deploy will be stable, and assurance that even if (and especially when) there is a problem in production, we can deploy a new version with confidence? Good news! The previously mentioned hardware abstraction and service encapsulation has made exactly that situation possible.

When you can place your software, dependencies, and configuration all inside of a container...well, that becomes an isolated unit of deployment. That means that you can spin that container down and spin up the previous version as quickly as someone can type the commands. Even better, what if we versioned our containers intelligently, and there existed a way to autonomously manage these containers? Wow - now we could just tell the automation system that if our new deployment fails to start, the old one should remain up! And if that new deployment succeeds, the system can reroute the connections to the successful deployment and remove them from the old one. It never even has to come down now.

That is powerful.

That is also just Production.

If we can swap out pieces of our software easily, and each of those pieces is, in and of itself, less complex – that is great. Take the example of the decoupled, containerized system above, and shift focus back towards the Development side of the house. When developers write their code, they can containerize it, and analyze the exact same environment as Production. No more “it works on my machine!” after a handy “git blame” or code review. Giving this power to the Development team means that they can see, test, and integrate their software as they go. If you duplicate your Production environment and call it “Staging”, you can make a change to Widget A and deploy it alongside the same services that it would

interact with on the Production system. If you automate this to happen (either organically or as part of a gated system), that is even better. In fact, it is continuous. Continuously deploying and integrating the components as development happens. Now when something heads up to Production, the Development team has seen it in action in the same environment, and the Operations team has much more confidence in the stability of the deployment, and their ability to react if/when something goes wrong.

Don't get it twisted

Ok, so we have hit some key concepts so far. DevOps as an idea probably makes more sense, and one of the practices that helps to get us there is the concept of CI/CD. But keep in mind that these are concepts - maybe even cultures. A process of steps and a collection of technologies does not "DevOps" make. Those things are how you support DevOps, how you do DevOps. Drilling this home is one of the most important things that can be done. This is especially true for executives who see the benefit, but view DevOps like a product that can be purchased. A DevOps process, team, or culture must be crafted, curated, and maintained. It must be updated, and that's the beauty - it can be updated. It is a flexible philosophy - it's one that gets things done, and switches gears if what has been isn't working any longer. And at MTSI, we are ecstatic at this new initiative, and the potential for industry-wide disruption as we dust off the keyboards of the DoD and push forward into the future.

Playing Ball

Putting the Sec in DevOps

So now, in some dimly lit board room, someone reading this from a high-backed, premium leather chair leans forward and looks at the checkbook now laying open against the mahogany table. It is time to focus – you have my attention, it sounds great. But how do we get there? What is this white paper even about? So far, you just heard a verbose explanation of the concepts that the DoD has already said they want to move towards. And that is true. But the point is that the DoD wants to get there. And we know the way. We have been there, done that, and are doing it right now. MTSI has been guiding our customers towards their desired destination for over 25 years. We are committed to staying at the bleeding edge of technology and keeping our finger on the pulse of not just the DoD and our customers' needs, but also the commercial world. We do the research, we spend our own money to buy a machete and hack our way through the jungle so that when our customers need to go in that direction, we can confidently guide them through.

But wait - what about Security? This is a whitepaper about DevSecOps! Absolutely right, and it is a critical distinction. But you cannot get to DevSecOps without focusing on how Development and Operations come together. When securing an application, Development and Operations both have a huge role to play: code tested, vulnerabilities addressed, and exploits closed. Environments for deployment need to be analyzed, secured, and hardened. To get to secure DevOps - to DevSecOps - we need to make sure we have the two disparate ends brought together before looking at how to integrate security across the board because it relies on and impacts the entire solution, soup to nuts.

Focusing on automation

Security is a broad concept - so broad as to rob it of meaning without a dose of specificity. For our purposes, we will focus on how security weaves into each aspect of DevOps. Automation is a powerful tool for DevOps. So, too, for security; when compared with traditional software development and operations, automation of security functions can simultaneously reduce implementation time and increase the thoroughness of security policies. As with other forms of automation, part of the beauty is reducing the possibility of human error.

Security with respect to software development usually comes in the form of testing and vulnerability scanning. Similarly, on the operations side, vulnerability indexes show where critical flaws may be hiding after analyzing dependencies. There are also Security Technical Implementation Guides (STIGs) that exist for common operating systems and platforms to ensure minimal vulnerability. On this front, containerizing our application code shifts the buck a little bit. We still need to ensure that our deployment environment is secured and hardened, but we also need to make sure the same applies to the container that holds our application code. Automating these tasks removes much of the extra time from duplicating efforts, and actually provides us with a more stable environment. Imagine a scenario in which a critical application requires certain security rules to be relaxed, or even removed. Without those rules being enforced, the system has a higher risk associated with that particular attack vector - but without that hole, the application software cannot run. If we containerize the application, however, we may be able to get away with only loosening the security requirements inside of the container and keeping a full enforcing rule set on the deployment system. In this way, even if a security hole must exist to perform the mission, an extra layer of mitigating security is in the abstraction that the container has provided. Keeping a known vulnerability that an application cannot run without does not always have to compromise the system it runs on.

So how do we begin to integrate these security practices (STIGs, hardening, vulnerability and dependency scanning) into our DevOps process to really do DevSecOps? Well, the solution is actually pretty straightforward; that is to say, simple in principle, but complex in implementation.

The first and most obvious step that most DevOps processes in use take advantage of is automated unit testing for their application code. That is, use a tool that runs pre-defined and configured jobs to run when triggered by an event. In our case, that may be telling GitLab CI (the powerful and integrated CI/CD automation tool bundled with GitLab) to run our suite of unit tests on our application code whenever a developer makes a commit, or a merge occurs between branches. But - what guarantees that a successful unit test suite run on one machine will catch bugs that may come from mismatched packages or dependencies, as so often happens between different systems?

Again, enter containers! If our developers have code checked out and are writing it on their machine, they still commit and check in that code to the centralized repository. Remember, we deploy our software using containers now, so all our dependencies can be isolated, with the container's internal environment state guaranteed. A container is just an instantiation of a static image, which gives us the assurance that (unless there is human intervention), any two containers instantiated from the same image will contain identical environments, regardless of where they are run. We can apply this principle to give our testing the same environmental guarantees that we get when deploying. When a developer commits code, we can use the same base image (the image that we start with, before adding our own code or artifacts) to run the unit tests that we place the final build artifacts into. This lets us know that the environment that the code was tested in is the same as the deployment environment, giving our successful test runs a little bit more gravitas than a developer's machine spitting out a successful test report. Add a static code analysis software to mix, such as HP Fortify or SonarQube (or both!) and you have a great way to continuously monitor the security, health, and stability of your codebase in an automated way.

Consistency is key

The consistency of the environment within the container ensures the consistency of our vulnerability checks. A tool can analyze the packages and dependencies that are within the container and compare them against the Common Vulnerabilities and Exposures (CVE) index. Issues can be fixed, baked into the image, and propagated to any containers that use that image as a base. So, if you STIG, harden, and verify that your image has no (or rather, an acceptable amount) of vulnerabilities, then you know that all your deployed containers that come from that will share those traits. And to be doubly sure, in the event that subsequent images may have changes that negate this, you can then proceed to scan the resulting child images containing the application code again.

At this point, from a developer's standpoint, this is business as usual - they are writing code and committing it. At the other end, an image with their application spits out. An image that has been scanned for vulnerabilities, has one or more STIGs applied, that contains an application that has also been scanned for vulnerabilities and thoroughly unit tested in an identical environment to the one it is going to be deployed in. And nobody had to touch it!

Adding human-in-the-loop gating is a great way to enforce that security policies are being adhered to, but it is not a limitation of the technology; the entire process could be automated, regardless of whether or not it should. So far these aspects of security are leveraging technology that is almost ubiquitous across DevOps already - so adding them in is not so much "bolting on security" as it is making sure that the traditionally automated steps of build → test → verify → scan → conditionally deploy.

These extra security steps are not like wrapping a rope in a metal tube – that is an easy step to keep the rope from being cut, but it is also going to restrain its flexibility and interfere with the rope's purpose. This approach of adding security at key steps in the process is like taking the rope, and weaving steel cable into the rope itself. It will maintain its flexibility, still resist the cuts, and most importantly, retain its original function. The rope will be even more useful afterwards, not less.

What if we went so far as to implement this design within our own deployed microservices? If we had a securely maintained CA, or even multiple intermediate CAs for various environments, like development or staging? That would allow us to provide our applications with certificates and keys that would trust each other implicitly by way of the trust chain (both certificates were signed by the same CA), and allow TLS encrypted traffic between themselves. Even more interesting, what if the generation of those intermediate certificates, and the application certificate/key pairs were automated like the deployments? That would give us the ability to automatically redeploy the entire system, ALL of our microservices, with a fresh new set of cert/key pairs. Even more than that, the intermediate CA that signs them has been recreated, and the old one revoked. Now the new cert/key pairs will NOT trust the old ones, and TLS handshakes between applications with mismatched certs would be rejected.

This process can be automated to the point of a time interval. Certificates and keys are simply files that can be mounted to a predetermined location within a running container. Swapping out revoked certs and keys, while replacing them with a valid set, is all able to be handled within the orchestration platform. Envision an environment where a malicious insider plans to impersonate a valid and authorized service. With containers, a best practice for communication is to create a Software Defined Network (SDN) to keep the containers and their traffic isolated from that of the host machine. So, first, the insider needs to gain access to that network, which is managed by the orchestration platform. Assuming the insider can obtain a network lease and join, they now have access to the network and can communicate. But the cert/key pairs that are required for all communication with authorized services are missing - so they cannot communicate with services on that network, even after the hurdle of getting on the SDN in the first place. If they somehow were able to get a copy of an authorized cert/key pair, if the renewal/revocation cycle is set up, they have an extremely limited window of time in which to cause mayhem. Couple this system with strong Role Based Access Control (RBAC), and the likelihood of a user gaining access to unauthorized controls is further reduced. Additionally, most authentication providers provide real time monitoring and logging in the event of malicious activity.

Putting a bow on it

These hypotheticals have hopefully served to deepen the understanding of what DevOps is, what it is not, and how DevSecOps operates with the same philosophies with more stringent security woven in. It should have also illuminated some potential technologies to aid in the execution of a DevSecOps process and helped solidify what it would look like in the real world. I am happy to announce that at MTSI, this is not just a hypothetical - this process, along with the tools and technology that drive it, is what we are investing in. It is what the DoD needs, where we want to go, and so important to us that we invested our own dollars to launch an IRAD for our DevSecOps Infrastructure as Code Agile Pipeline (iCAP) in order to create a reusable, scalable DevSecOps technology stack that will help drive both our internal DevSecOps processes and those of our customers.

We focus on Infrastructure as Code (IaC), using Packer to define and configure our machine images (think VMs), and Terraform to provision them across various providers - from internal private clouds like VSphere and OpenStack to external cloud providers like Azure and AWS. On top of our infrastructure will sit OpenShift, the RedHat Kubernetes distribution and orchestration platform, which will manage all of our tools and services. Configuration as Code (CaC) will go hand in hand with our IaC definitions, such that the entire toolset will be distributable as code.

Being able to clone repositories and spin up a customizable, configurable DevSecOps toolchain will provide customers a Day 1 start and avoid the often months-long start up time to provision infrastructure, set up the tools, and begin working on processes. As stated earlier, DevOps (and by extension DevSecOps) is a process that is enabled by technology. Having these tools up and running is just opening the door - one still has to walk through it. But that is much easier than having to bust down a wall yourself.

Engineers designing this system have a diverse background working with Microservice-Based, Event Driven Architectures (EDA), Infrastructure as Code, Configuration as Code, and led numerous development teams using Agile methodologies such as Kanban and SCRUM. From DARPA's Hallmark Software Testbed program, SMC's FORGE platform, to Restricted programs focused on pivoting away from the waterfall-style development paradigm, our team is ready and capable of bringing us to the bleeding edge.

We pay the startup fees so that our customers can see the results and test-drive our solutions without having to foot the bill for it. They can spend more time and money on completing the mission rather than having to reinvent the wheel every time.