如何将 InfoSec、Compliance 集成到持续交付流水线中

Setup

“负责安全审查的部门不会让我们这么做的”，这通常是组织推进DevOps转型中遭遇的最棘手的问题，尤其对于一些各种流程相当健全的巨无霸公司。本人并不是infosec与complance方面的专家，但有幸在推进企业内部devops转型的过程中，尤其在搭建持续交付流水线的过程中被安全部门无情的challenge过二次，于是乎对一些应对安全审计的流程以及所需准备的材料有了一些了解，我把经历的一切以及应对策略记录了下来分享给大家，希望大家可以得到一些启发，欢迎拍砖。

痛点

一个公司的信息安全部门往往被公司赋予至高无上的权力，可以直接决定你的新服务或功能是否可以按你的节奏上线。《凤凰项目》一书中描述的那个黑色文件夹里面定义了多达数十页的信息安全的相关流程，想想都会头大，如果你没有那么走运，身边没有一个像Eric【注1】那样的人物罩着，我劝你还是小心不要招惹他们。一旦被他们盯上你就麻烦了，不但你的新功能被block到那，而且你需要被迫阅读大量的安全部门提供的合规文档，然后就是准备各种材料等待安全部门审查，接下来就是无休止的meeting, 你懂的，如果你服务于一家跨国企业，那么你就惨了，负责安全审计的同事一般是个老美或者生活在美帝的老印，所以不但要忍受时差熬夜开会，回答各种稀奇古怪的问题，有时还要忍受老印滔滔不绝的讲个不停，其实你根本不知道他在说什么，最后他还会丢给你一句“make sense？”。刚好我遇到的是后者，而且是两次，可见我的心智已经足够成熟与坚挺。。。汗！

问题来了

言归正传，总结一下他们关心的无非就是下面几个问题：

1. 如何保证你的新功能的代码有没有引入漏洞？
2. 如何保证你的新功能所引用的第三方代码库没有引入漏洞？
3. 迁出源码到CI server上后，如何保证代码的安全，（a. 只有开发人员可以提交代码 b. 代码不会被泄漏）
4. 如何审计每一次部署？
5. 如何做到变更管理？

6. pipeline的权限管理

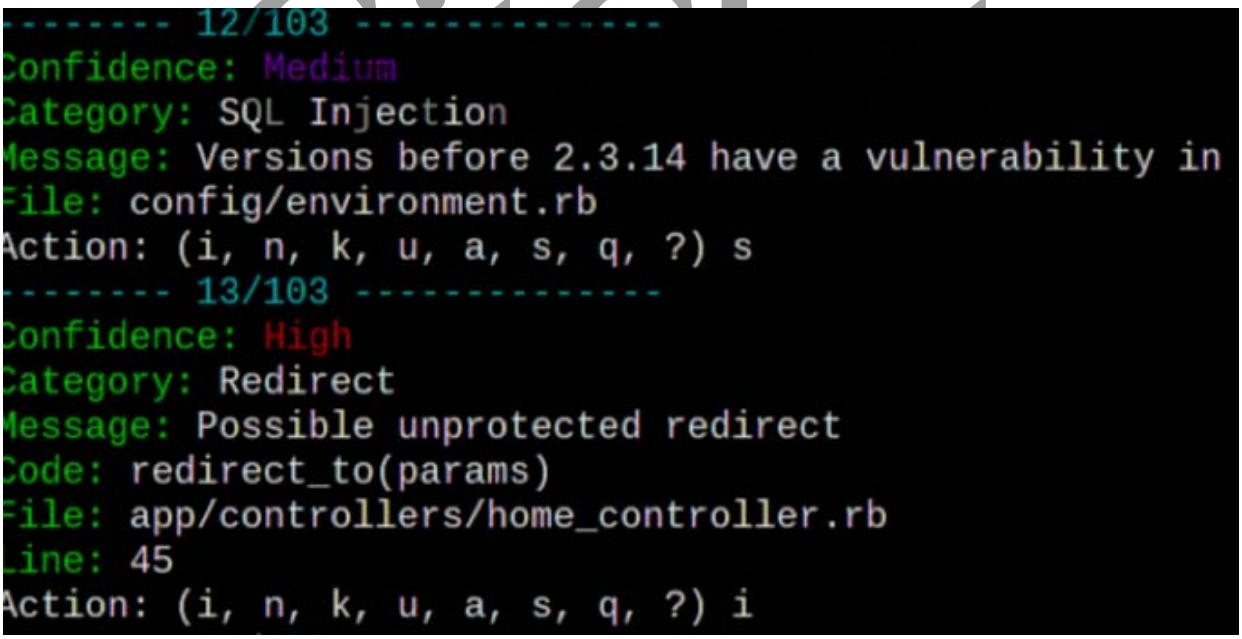
如果你还遇到其他的问题，欢迎来Chat！

应对策略

“如何保证你的新功能的代码有没有引入漏洞？”

静态漏洞扫描

在我们的交付流水线中通常会有一个步骤叫做静态代码扫描，来帮助我们分析源代码，找出代码存在的缺陷：潜在的bug，未使用的代码，复杂的表达式，重复的代码等。同样对于安全性安面的漏洞，也可以通过源码分析来解决部分问题，由于本人这些年对ruby技术栈的痴迷，下面我们来看一个基于ruby on rails的工具类项目通过引入brakeman来实在静态代码扫描来分析安全隐患的例子，如下图所示像SQL Injection, Redirect之类的漏洞会及时的被detect到，能过brakeman的输出结果，build脚本可以按照项目自身的要求来判断是否让build失败。关于brakeman详细信息请看这里https://github.com/presidentbeef/brakeman，我google了top10的扫描工具以供大家参考。



另外，对于企业中的安全扫描，大部分情况下安全部门会提供一些library来帮助扫描你代码中的常规漏洞，或者安全部门会要求开放你的代码仓库，由他们的人来完全安全的检查。我接触到的做的比较好的方式是这样，安全部门会以API的形式将安全服务暴露给团队，这种情况下你需要做的只是实现一些自动化安全测试来调用安全部门的API，然后通过判断API的结果来判断是否通过安全扫描，最后将这些集成到你的交付流水线就OK了。

动态安全监控

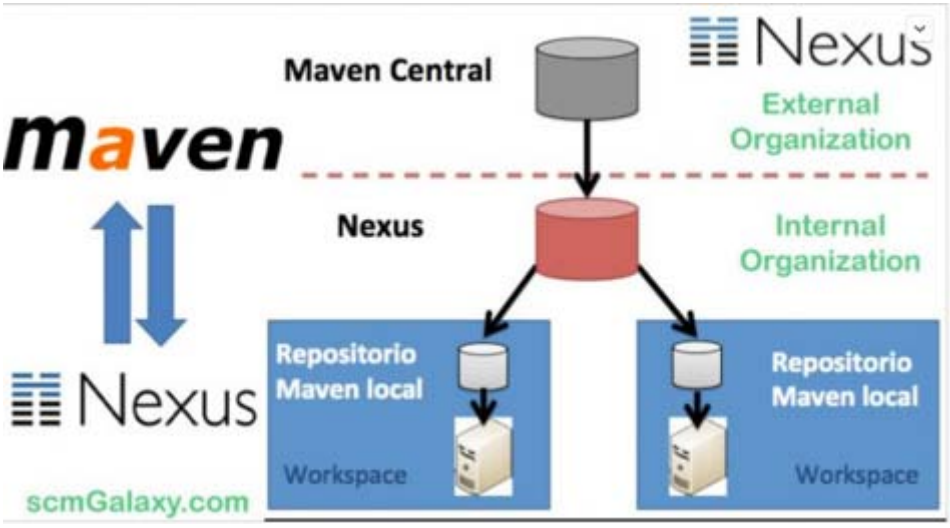
有一些漏洞是没有办法能过静态代码扫描分析出来的，例如你的服务有可能非常的容易被黑客攻击而导致瘫痪，这种情况下就只能由线上的监控系统来monitoring了,监控系统会按照团队预制的行为来评判线上系统，一旦发现线上服务有不符合预期的行为，那么监控系统会尽可能全的保存现场所有证据并发出告警，这种方式很容易发现服务有没有被植入malicious code或malicious app攻击。能否及时有效的发现线上服务的故障取决于定义的metrics是否能够覆盖相关问题。

本着没有比较就没有伤害的目的，我们来看一下facebook的监控系统，几年来已经积累了100多万的metrics来保证提供的服务正常运行，这还是2年前的数字。我之前服务的公司，主要系统的metrics量也有几十万个，所以要鼓励团队收集尽量多的metircs,如果一时做不到“足够多”，那么至少要保证遇到一次问题，完成相关问题的监控metrics,从而保证不被同样的问题愚弄第二次。然后就是需要整理一下相关联的关键指标，把关系比较紧密的做成dashborad以便于综合分析比较，同时也方便查看历史数据与评估未来趋势。推荐的工具是grafana,如下图所示这是一个network相关指标的dashboard，如果我们发现在当前的连接数突然大幅增加或者In e Out两条曲线差值过大，可能你就要去查看一下到底发生了什么事情。

如果你所在的公司比较高大上，能够提供self-service的线上安全服务以及监控框架，那么你所需要做的只是实现相关的metrics，安全部门的监控系统会帮你时时监测，发现在漏洞当然也会通知你。

“如何保证你的新功能所引用的第三方代码库没有引入漏洞？”

软件开发不可能从零做起，开发过程中肯定会用到一些第三方的代码库来帮助我们加快开发进度，但某一些第三方的代码库确实存在一些漏洞，像openssl早期版本也被攻击者利用过。这个问题对于在企业中安全审查就比较简单了，安全部门通常会有一个很长的黑名单，清晰的标注了文件名与版本号，只要不用黑名单中的包就可以轻松过关。对于一些比较“正规”的公司来说更是容易，往往这些公司都会拥有自己的私有仓库，私有仓库中的包都是经过安全部门鉴定过的，产品代码中引用的包只能从私有仓库中下载就可以了，从根本上避免了第三方库的安全问题。下图是一个maven与私有仓库交互的示意图：



“会迁出源码到CI server上，如何保证代码的安全？（a. 只有开发人员可以提交代码；b. 代码不会被泄漏）”

关于这个问题，首先我们要保证持续交付流水线为了迁出源代码所用到functional ID对版本仓库只有只读权限，这样的话即使CI server或者pipeline被黑，也会避免源代码遭到破坏的风险，另外需要注意的是build完成后要及时清理CI server的workspace来保证源码不在server上长时间保存。
接下来就是保证server的安全性，关于sever的安全管理很多实践，我们常用的就是每隔三个月更改一次密码。

“如何审计每一次部署？”

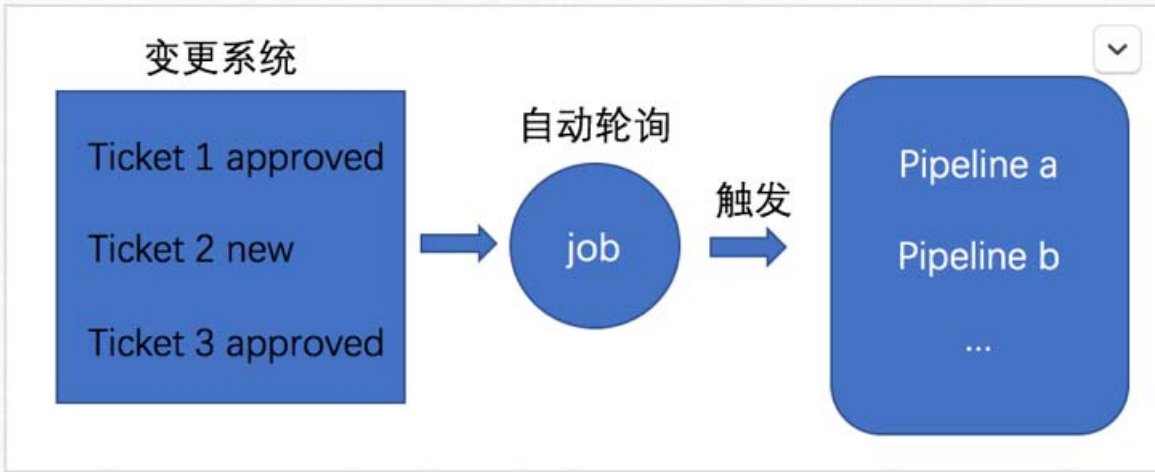
答案很简单，就是log everything. 安全部门关注的是 — 谁？在什么时候？部署了哪个版本的code？为了fix什么样的问题或发布什么新功能？通常情况下，每次代码发布我们都会在变更系统上提交一个ticket, 用来描述我需要使用哪个版本的代码以及简单描述该版本的新功能或fix的问题，一方面用于自动部署工具根据这个ticket的描述抓取相关版本的文件，另一方面也可以做为版本变更的依据，一旦部署完成后线上系统出现故障，可以迅速定位是哪个版本有问题，而且通过查看变更系统的历史可以知晓上一次可以正常运行的版本，一旦发现总是可以做到快速的回滚。另外就是pipeline本身的每一个阶段 — build，test，deploy…，只要有产出的log都尽量在ticket上加一个comment, 方便跟踪整个过程，当然如何你已经搭建好集中化的日志收集管理服务，可以把所有相关log收集到那里，然后跟变更系统的ticket做一个关联，这样负责audit的同事就可以很容易的获得他想要的所有信息了。推荐的集中式日志管理工具当然就是ELK或者graylog。下图是我们的一个变更ticket的log信息。

“如何做变更管理？”

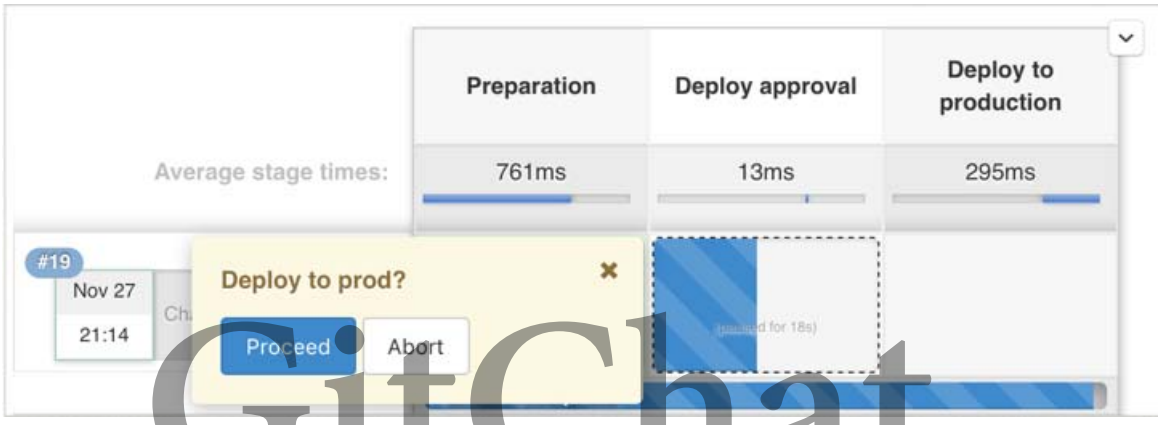
通常情况下我们平时工作中遇到的变更大概可以分成以下三类：

1. 常规变更操作，这种情况下的变更一般是周期性的，每周，每月或更长的时间周期做一次，不会有功能的变更主要是处理一些数据，比如更新季度报表等。这种变更是不需要任何的approve的，到了预定时间就可以被运行，当然往往这种操作是要交给我们的pipeline去自动化完成的，可以简单的用系统或者pipeline的scheule功能定期完成相关部署，但一定要做好日志收集的工作以便安全部门的同事audit使用。

2. 标准变更，这种变更往往是需要部署新的代码变更到生产环境的，这种情况下往往在一些流程超级健全的公司是需要某负责人approve之后才能执行，虽然我们的代码通过了一系统的单元测试以及在各个testing环境自动化测，而且有相当完备的监控系统与回滚机制，但往往还是很难绕开approve这一环节。抗争的结果只能从长计议，待慢慢建立信任后逐步取缔。当然在没有取缔前，通常的做法就是在变更系统中增加approve功能，开发人员需要先定义好部署的源代码版本，提供一系统证据来证明该版本的代码已经被review过并通过所有测试并ticket提交，审批通过后才可以由pipeline部署上线，下面是我们流程的示意图，有一个job去轮询变更系统，然后将approve的ticket转发给pipeline完成部署：



也有同学将approve的动作放到了pipeline中，如果下图所示，在pipeline加了一个approve的stage:



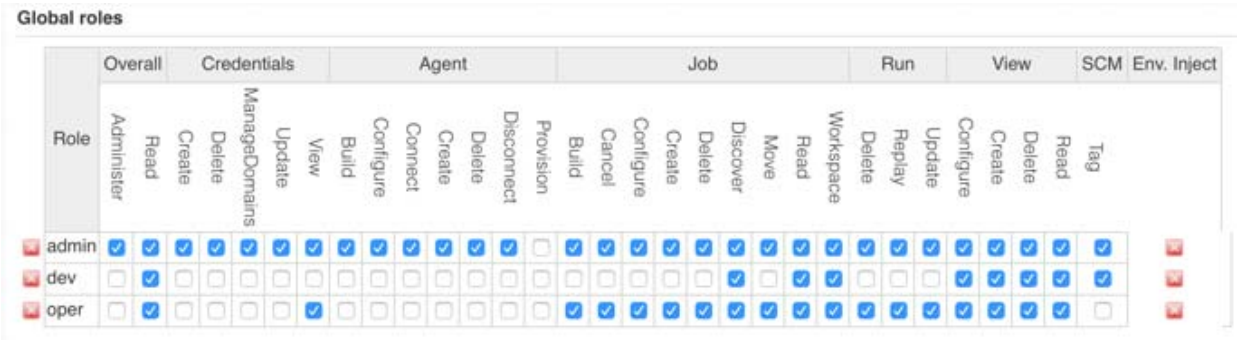
我不太推荐这种做法，虽说是把approve的操作集成到了pipeline，但感觉没有什么意义。变更负责人还是要通过查看变更系统的ticket来评估版本可否上线，之后还需要登陆jenkins找到相关的build，点一下approve来完成审批，反而增加了复杂度。另外，当pipeline进入等待审批阶段时，pipeine会hang到那里等待，如果这样的deploy很多，那么你的jenkins Idle将会被占满，其他build或deploy将被放入queue中等待.

针对这种情况，我们也做了另外一种偿试，我们写了一些自动化脚本去模拟审批人所做的动作，也就是利用code来实现审批功能，目前这种方式我们在小范围应用，也慢慢取得了manager的信任，随着模拟审批的code功能越来越强大，会逐渐扩大应用的范围。

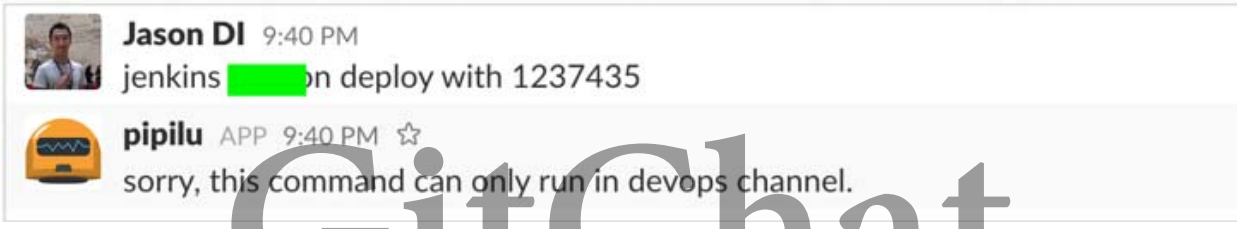
3. 紧急变更，这种变更严格意义讲是不是需要审批的，可以紧急上线，越早fix问题就会越早的止损，虽说这种情况极少发生，一旦不幸发生了，那么需要在变更系统中记录下详细的信息描述以及解决方案。如果你所在的团队，运维体系够健全，最好还要事故现场的所能收集到的信息以及所有操作的时间线存放放到defect系统中，为了日后做根因分析，增加监控metrics等工作提供依据。

“pipeline的权限管理”

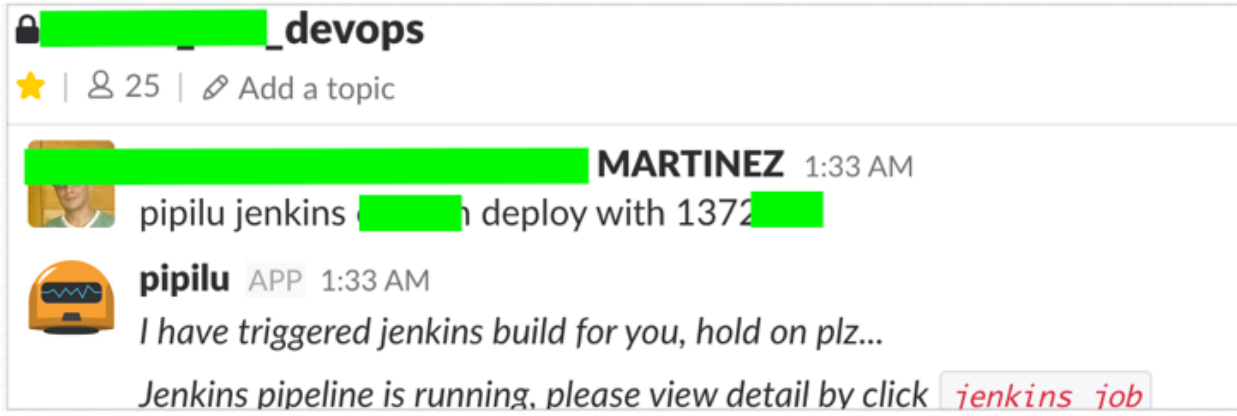
最后就是关于pipeine的权限管理问题，以jenkins为例，jenkins提供了完整的权限管理机制，我们可以通过两种不同层次的Role来管理团队成员对pipeline的管理与使用权限。下图是我们的权限管理示例，项目角色与全局角色的区别就是，项目角色只能管理项目，没有管理jenkins的权限配置



当然如果你的pipeine有不同的入口，例如你的团队引入了聊天机器人来帮助团队部署应用，那也一定要明确机器人什么情况下才可以影响部署请求.我的做法是只会让机器人在特定的channel响应特定人的部署请求，这样方便追踪与管理。下图示一个不响应部署请求：



在特定channel中响应特定同事的部署请求：



谢谢你坚持读完，欢迎大家多交流！

注1：《凤凰项目》这本书主要讲的是Bill带领的团队在Eric协助下如何改进IT运维、促进业务提升，并最终使得公司起死回生的故事。过程中Eric做为转型顾问帮助Bill解决了很多价值流，反馈环以及信息安全的问题。