DREADful

Article08/14/2007

Both the STRIDE and DREAD systems Michael and I documented in Writing Secure Code have been criticized quite a bit. Neither of them were developed with any real academic rigor, and from a scientific standpoint, neither of them tend to hold up very well. STRIDE has a number of cross-correlations – for example, escalation of privilege (E) tends to imply spoofing and loss of non-repudiation, and could imply tampering, information disclosure, and denial of service. Ouch – every vulnerability category we had, all in one bundle. The fact that it isn't a rigorous classification doesn't mean that it isn't useful – it is useful in helping people focus the debate about what to do about some specific problem. Outside of the ivory towers of academia, there's a lot of very useful things that don't pass muster with a strict academic standard.

DREAD is something that came up out of the Visual Studio security push. A bit of trivia – that was actually our first security push, and preceded the Windows security push for Win2k3. I believe that DREAD is actually a fairly reasonable model, and that the biggest problem is how we implemented coming up with an overall score. Let's review the components:

D = damage – how bad is the problem? This has to be judged in the context of your app. For a server app, any sort of crash is probably a bad thing, whereas if notepad crashes non-exploitably, that's an anklebiter attack, not really a vuln.

R = reliability – how reliable is the attack? Does it hit every time? Or does it not work reliably? Note that this is orthogonal to exploitability, and says nothing about the preconditions needed to launch the attack, just how well it works once you trigger it.

E = exploitability – how much work is it to launch the attack? Worst case is anonymous access, and it can be scripted. Such things can be turned into worms. Best case is that it requires a fairly high level of access, such as authenticating as user, or it requires a special configuration, or maybe an information leak. Maybe you have to guess a certain web page's path and name, and then launch it. I've seen exploits that took a penetration tester an hour or more to set up by hand.

A = affected users – pretty self-explanatory – all the users? Or just some of them?

D = discoverability – perhaps the most controversial component, but good information to have. Something that's highly discoverable is publicly known, or very similar to something that is publicly known. Low discoverability is that it takes intimate knowledge of the internal workings of your app to sort out. This one is really risky, since what you thought was hard to sort out could get published on BUGTRAQ tomorrow. My advice is to not take comfort if something is low discoverability, but to get really alarmed if it is high. More on this in a moment.

If we look at the five components, we see that none of these are highly correlated – one of them does not imply the other. This means we have independent factors, which is one of the strongest criteria for a solid model. Thus our task is to figure out how to properly weight the inputs. In WSC, we told you to rate them from 1-10, add them up, and divide by 5. If we apply some obvious tests, we find that a damage of 1, and all other factors 10 (a well known nuisance, e.g., pop-ups) gets weighted the same as a discoverability of 1 and everything else 10 (hard to sort out, but causes the heat death of the universe). This is an obvious malfunction.

Next, what's the difference between discoverability of 6 and 7? Who the heck knows? I don't. If we're going to have big error bars, let's simplify matters and drop back to high, medium and low. We can all fairly easily sort out 3 categories that pertain to our app.

As I thought about the overall problem, another aspect is that our factors group into two classes. Damage, reliability and affected users all group together to tell us how bad something is, and tends to remain static with respect to the other two factors, so we can safely say that:

Severity = f(Da, R, A)

The remaining two factors each vary a lot depending on situation. Say we had a vuln that required user-level remote access. This is next to useless against home users, since most of them have blank admin passwords that can't be used from the network, and/or they have firewalls up. However, on a corporate network, this could be a really easy bar to hit. This implies that exploitability is very situational, and may vary a lot from one customer to the next. Discoverability is a SWAG at best, and can change overnight. Let's then use these factors to add a priority boost if they're high.

Now that everything was down to 3 levels, I managed to put it all in a spreadsheet, and see what worked. Here's what I came up with:

Base severity (1-5) = Damage + f(R, A)

This weights heavily for damage, which is as it should be. Implies that if damage is high, severity cannot be less than 3 of 5. I then said that:

```
If R + A > 4 (both high, or one high and the other at least medium), add 2
else if R + A > 3 (one high, or both medium) add 1
else add 0
```

Reviewing the results, this seemed to make sense, and work fairly well. I then mapped the resulting 5 categories into the priority as established by exploitability and discoverability into 9 buckets:

Expand table

Fix By	Value
Won't fix	1
Consider next rel.	2
Next release	3
Consider SP	4
SP	5
Consider bulletin	6
Bulletin	7
Consider high pri	8
High Priority Bulletin	9

This means that if severity is 5, that even if the priority factors are low, it wouldn't be less than a SP fix. Seems to make sense. To establish priority, I used this approach:

```
If E = high, Di = high, add 4
else if one is high, other medium, add 3
else if one is high, other low, add 2
else if both medium, add 1
else add 0
```

This also means that a well-known, highly exploitable nuisance still ought to get fixed in a SP.

Some caveats – we're NOT using this internally very much. This is NOT how MSRC does things. This is just something I sorted out on my own, and hope it is helpful to you. Warning! Do NOT apply this system, or any other system, without THINKING about it. (One of my favorite professors, who was from Sri Lanka, often said "you have to THINK about it" – he's right) This system may or may not help you arrive at the right conclusion, and if it does not, consider worth what you paid to get it, which is zero. If you think you can improve it, that's great, and if you post a helpful comment (as opposed to the "Nice" or "Interesting" spam I seem to be beleaguered with), I'll publish it.

I have a spreadsheet that works up the 27 cases that lead to the severity score, and the 45 cases that factor in priority, and it seems to work well. I have to really work to find things that don't seem to make sense, so I think this is a lot better than averaging things.

Comments

- **Anonymous**
  August 13, 2007
  So, what *are* you using at MSFT? Still some DREAD based model (perhaps with different numbers), or something totally different? If it's not "super corporate secret", I'm sure quite a few readers would like to know, even if it's only the basics or "not quite ready for prime-time"/ [dcl] This varies from one group to another. For example, some of the operational groups use a modified DREAD system with different weightings, which is part of what gave me the idea for this system. Office has been using the old system, but recognize that we need to discuss it, and the complaints around that is one of the reasons for coming up with this system. This system isn't in wide use yet. I can't think of anything about our SDL process that's a secret. We want you to be able to ship software as securely as we do. If there's something we're keeping as internal, it's usually because it isn't supportable.

- **Anonymous**
  August 14, 2007
  My experience with DREAD and STRIDE is that the severities make the most sense when calibrated against other security issues. When looking for vulnerabilities, one

reviews all the code in a particular area, not just one item. The analysis should always be:

1. Review code, noting possible issues.
2. Review issues, assigning STRIDE/DREAD scores. Finish 1, then start 2 (it's ok to contiue to update 1 while doing 2). What you wind up with is a list of issues ranked relative to each other, and this allows you to strategically allocate resources to issues. I think that looking at an individual issue in a vacuum just doesn't work. Sorry for being so verbose... [dcl] Sure, that completely makes sense from an audit or code review standpoint, but what about when you're triaging an internal or external find?

- **Anonymous**
  August 14, 2007
  The comment has been removed

- **Anonymous**
  August 14, 2007
  When triaging a single item, we have a bunch of things we've already done when doing the initial DREAD analysis. So, we just rank the new issue against the existing items.

- **Anonymous**
  September 04, 2007
  As has been mentioned elsewhere , when we're threat modeling at Microsoft we classify threats using the

- **Anonymous**
  September 04, 2007
  As has been mentioned elsewhere , when we&#39;re threat modeling at Microsoft we classify threats using

- **Anonymous**
  May 31, 2008
  Both the STRIDE and DREAD systems Michael and I documented in Writing Secure Code have been criticized quite a bit. Neither of them were developed with any real academic rigor, and from a scientific standpoint, neither of them tend to hold up very well

- **Anonymous**

June 06, 2008
Both the STRIDE and DREAD systems Michael and I documented in Writing Secure Code have been criticized quite a bit. Neither of them were developed with any real academic rigor, and from a scientific standpoint, neither of them tend to hold up very well

- **Anonymous**
  April 03, 2017
  David - a brilliant post though I discovered it only now. Could you be kind enough to share your spreadsheet with me - jagan.vaman@gmail.comI am a Cybersecurity and GRC consultant so I will use it in my projects. So if you are OK with this - kindly share.Highly appreciate your support. Best regardsJagan Vamanwww.cymax.in