

Writing secure and reliable online game services

for fun & profit

by Patrick Wyatt

Use "Yanone Kaffeesatz" TrueType font: http://www.yanone.de/cgi-bin/download.pl?file=kaffeesatzfont_ttf ("regular" size)

Robust

services & software

I'm going to talk about engineering robust software, but because I only have an hour, most of what I'm doing will be *evangelizing* robust software. But that's just as important, because it is massively overlooked in massively multiplayer online games.

First, I'd like to talk about robustness by using an example from outside software development and the game industry.

Several weeks before this presentation, I was buying chocolates for my wife for Valentine's day. A cashier at the chocolatier helped me select a plate full of tasty yummys, set it on the counter in front of the register, and went in search of a box to hold all of them. Another cashier similarly helped my friend, rang up his chocolates, and pressed the "cash/tender" button. This caused the register cash-drawer to pop open, shoving my plate of carefully selected chocolates off the counter and onto the floor. Needless to say the cashier was quite embarrassed, and my cashier was angry, and the chocolate store was out the cost-of-goods for those chocolates (assuming no "recycling" happened after I left the store).

In a larger franchised organization, we might imagine a post-mortem meeting where the staff analyzed root causes, drew up a plan to encourage cashiers not to place chocolates in front of the register, and sent the new policy to headquarters, which then distributed a memo to all franchisees to add to the already-too-large three-ring-binder policy manuals. You might laugh, but if you've worked in a franchise

organization and seen the three-ring binders, you'll know this to be how these organizations function. As Neal Stephenson pointed out in *Snow Crash*, three-ring binders contain the entire operational blueprint for franchise organizations – the “franchise DNA” if you will.

This is total foolishness!

If I was in charge, I'd either bolt the cash register to the counter so that there wasn't enough space for the plate to fit in front of the cash-drawer, or I'd build the counters to be the exact width of the register.

MMO game developers builders need to be aware of similar traps that they set for the Customer Support and Network Operations teams when they build software. It's easy to build software that works properly for the development team, but it quite hard to operate in real world situations.

In this talk I'll endeavor to discuss how to build robust systems to avoid these types of operational issues.

But before I begin, I'd like to explain **why** robustness is so important.

When early-adopter gamers start playing your game, the quality of the experience they have is going to determine whether they tell their friends to buy or avoid your game. This has a massive influence on your likelihood of success. Your sales are directly related to whether you win or lose these critical players, because each player who cannot play your game due will tell some of their friends not to buy it, and each player who enjoys your game will tell their friends to buy it. So the adoption curve for your game is a polynomial factor of the enjoyment of early adopters, not linear!

Lead/network programmer:

Warcraft, Diablo, Starcraft, battle.net

lead programmer: Guild Wars file streaming

lead programmer: Guild Wars server backend

technical lead: TERA account & billing platform

Why are we here?



What expertise do I bring to the table in these areas? Well, I was either prescient enough or lucky enough (more likely the latter) to start building multiplayer games in 1993, when players were tolerant of game design mistakes. So I made lots of them and learned a bit about developing robust, reliable and secure online games, and I'm going to share some of what I've learned so that you don't have to make the same mistakes I did – you can make new ones!

Linux (epoll)

```
int eventcnt = epoll_wait (  
    backend_fd,  
    epoll_events,  
    epoll_eventmax,  
    timeout);  
if (expect_false(eventcnt < 0)) {  
    if (errno != EINTR)
```

Windows (iocp)

```
rv = GetQueuedCompletionStatus(  
    _pr_completion_port,  
    &bytes,  
    &key,  
    &olp,  
    timeout);  
if (rv == 0 && olp == NULL) {
```

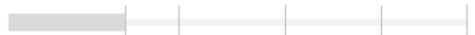
Why are we here?

Let's dive into code: *nix programmers follow LibEv code on the left, Window programmers follow IO completion ports code on the right.

Well, I think this code speaks for itself, so if everyone understands the basics of the inner event-dispatch loop, let's move on! ☺

Too low level!

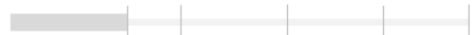
Why are we here?



Okay, talking in detail about code in a spoken presentation is more-or-less impossible. Let's zoom the map-scale slider back several notches; what are we really trying to solve for?

Reliability

Why are we here?



Today I'll talk about reliability, which is a key factor in players being able to enjoy your game. In the bad old days when we built single-player games, each game crash would only take down one player. But now, when a game crash occurs, it can kick 5000 players offline simultaneously. That means that, in order to maintain the same frequency of crashes that we had in single-player games, we effectively have to write software that is 5000 times more reliable than in the past. If that's not daunting, I don't know what is. That's why reliability is so critical.

Reliability

Security

Why are we here?



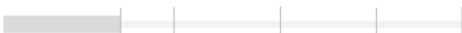
Since I expect I'm primarily speaking to developers who are building persistent world online games, security is also an important factor that needs to be discussed. There are legions of people with bad intent who try to ruin online games, either because they're griefers who achieve personal enjoyment by destroying the game experience, or because they're professionals who are paid to exploit the game.

Reliability

Security

Scalability

Why are we here?



And finally, if we've managed to build a game that players love, we can expect to have many players joining our game, meaning we'll have to grow our game service rapidly, so scalability is yet another key topic to discuss...

Reliability

Security

Scalability

Why are we here?



But this is only a one hour talk; I don't have time to discuss scalability in addition to the other topics. I'd encourage you to check out highscalability.com because it contains many great articles about how other companies have scaled to truly massive numbers of users.

Reliabilit y

Let's talk about building reliable software first.

[note: word-break is intentional]

Send(&important_msg) ... time passes ... Receive(&reply)

What could go wrong?



Let's assume we have network transport and protocol libraries... let's write code. Fundamentally all interesting game stuff comes down to this pattern – the client sends a message to the server, and the server responds. Or similarly, one server sends a message to another server and receives a reply. So this is an important pattern in development. Let's ask ourselves the question: what could go wrong?

Hardware failure

What could go wrong?



Well, we have to plan for the servers breaking down, right?

Hardware failure

fat-fingered a server

What could go wrong?



Well, hardware failure does happen, but 80% of downtime is caused by human error, so we need to plan for it by writing services that are easy to administer.

One incident that comes to mind is from a time when I was the lead battle.net programmer. (Actually, at the time I was the only programmer, designer, business-person, dishwasher and janitor for the project).

I came into the office in the morning and learned that no one whose name (“handle”) that started with the “d” could log into battle.net. I was able to quickly ascertain that the name-filtering/chat-filtering algorithm was blocking words starting with the letter “d”. It turns out that the Customer Support lead, who had administrative permissions to the battle.net servers, had attempted to block an obnoxious player whose name was something like d1oggone, so he added that name to the list of blocked words.

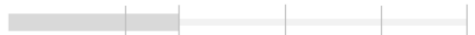
Unfortunately, the original programmer of battle.net, who had written the chat-filtering system, an otherwise spectacular programmer, hadn’t written much in the way of error-handling for the chat-filtering subsystem. Whenever the configuration-file parser read a “bad word” that needed to be filtered, it assumed that the word would only be made up of letters. When it encountered any non-alphabetic character it would simply stop parsing and terminate the word. So when it read d1oggone, it substituted “d” and used that in the filtering algorithm.

Much software, if not written quite this poorly, contains similar gotchas. And consequently the operations team ends up unintentionally bollixing game servers because they were led into a trap by the development team.

We can do better!

Network congestion

What could go wrong?



Another common complaint by players is that games are laggy...

Network congestion

Bogus network code

What could go wrong?



But y'know what? For the most part it's not the Internet's fault. The Internet is substantially faster and more reliable than it was in previous years. For the most part the problem is that development teams don't do much to verify that their code works properly under real latency, packet-drop-rate and bandwidth conditions. Most teams are writing and testing code on their local area network, which has massive bandwidth and virtually no latency or packet drop.

I got a late-night call from a former colleague at a company called Click Entertainment (unsurprisingly no longer in business; you'll soon understand why) just after they launched their game. They had written a Diablo-clone, and it turned out that the game, which had been thoroughly tested for network play on a LAN, didn't work over the Internet. Both the development team and the publisher (Sierra Entertainment) QA team hadn't ever actually bothered to test Internet play, even though that was the primary online venue. And so of course it didn't work properly and players were furious. Whoops! Turns out they had a bug that was actually easy to diagnose and suggest a fix over the phone, leading to a patch. But all the pain could have been avoided by decent testing.

For Guild Wars, when we built out the first datacenter for testing, we colocated the servers in Los Angeles, while our dev team was in the Seattle area. This meant that our entire development team played the game in similar fashion to the way our players would *four years later when we actually released the game*. Incidentally, we

also gave all the developers “minspec” machines (800 MHz Pentium III with 512MB RAM – blech!) so that we would build a game that played well on that type of computer.

Another great testing methodology is to use latency simulators that can control bandwidth, packet-drop and packet round-trip-time so that the development team can build code that handles typical (or worse) network conditions.

Socket disconnection

What could go wrong?

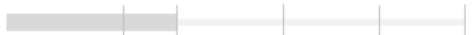


Another player complaint is that “the Internet” causes game-server disconnects, and many times it does...

Socket disconnection

crashy game code

What could go wrong?



... but other common causes of socket-disconnection are that the game server doesn't handle network timeouts or network packet-buffering, or even crashes a lot.

Plan for failure

What could go wrong?



When you get down to the packet level, the Internet is a scary place. If your program doesn't handle these issues then your users will get to discover them. And by extension, so will your Customer Support team, and consequently your ongoing sales & players retention will be negatively affected. So build your code so that it handles common failure cases instead of being surprised by them.

You should expect that some percentage of your players will lose connectivity to the server – usually at a critical time in battle while they're completing a multi-hour-long epic quest. Build in a system to enable them to immediately reconnect to the game from the *start* of your development cycle instead of waiting until after launch and discovering how frequently it happens (... like we did in Guild Wars... boo hoo!).

Reliable Transactions

In the next part of the talk I'm going to spend a lot of time understanding common reliability problems in backend services, where ultimately all of the interesting things going on are transactions. While this might sound like boring stuff that's more important for banking software, it's the same stuff that your game uses to persist player data, and is consequently critical to the success of your endeavor.

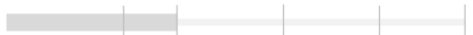
UPDATE items

SET gold = gold + @gift WHERE id = @receiver

UPDATE items

SET gold = gold - @gift WHERE id = @giver

What could go wrong?



Let's start with an easy one before we dive into the complicated cases.

Here's a basic transaction that's designed to enable one player (me) to give another player (you) some gold. 'Cause I'm just that nice a guy.

What could go wrong?

This is **two** transactions

UPDATE items

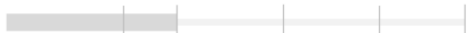
SET gold = gold + @gift WHERE id = @receiver

UPDATE items

SET gold = gold - @gift WHERE id = @giver

***in SQL Server**

What could go wrong?



What could go wrong?!? Well, for a start this innocuous-looking SQL code is actually two transactions (in Microsoft SQL Server), not one.

That means that there's a remote (but non-zero) possibility that the first transaction can succeed and the second can fail. And consequently the gift-receiver can get extra gold without the giver losing gold. This problem exists in many games, and when it is discovered by hackers they can duplicate gold and destroy the game economy if the problem is not immediately corrected. I'm looking at **you**, Everquest.

It is pretty trivial to switch the order of operations so that this problem is no longer exploitable, but now what happens when the second part isn't completed is that a player **loses** gold when something goes wrong. So that's no good!

This is **one** transaction

begin transaction

UPDATE items

SET gold = gold + @gift WHERE id = @receiver

UPDATE items

SET gold = gold - @gift WHERE id = @giver

commit transaction

What could go wrong?



Well, there is a simple solution to this problem; let's wrap the whole thing in a transaction to make sure that the two operations are always conjoined; either they both happen or neither does.

Well... that was trivial.

But this basic problem is actually made more complicated as we move to multi-server or multi-datacenter environments. As we'll see shortly...

Error: Double-tap transactions

What could go wrong?



But to solve problems we're going to need to understand all of them, so here's another basic one that occurs in far too many games (and web services).

User: <clicks buy>

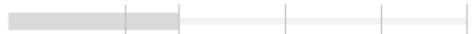
What could go wrong?



Hey, this is great; one of our users likes our game enough to buy something like an in-game item!

User: <clicks buy>
Hey: why so long?!?

What could go wrong?



Unfortunately, because we didn't test our game services under Internet latency and high-load conditions, the purchase result isn't instantaneous and the user gets frustrated...

User: <clicks buy>
Hey: why so long?!?
<clicks buy again>

What could go wrong?



So they click buy again... 14 times! And what happens? They end up with fourteen successful purchase transactions, and are of course pissed-off when they get their next credit-card statement (not to mention the frustration *during* the purchases).

So what can we do...?

Web server solution: redirect after POST

What could go wrong?



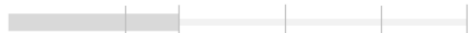
In the world of web-servers, when a user clicks “buy” on a web page, the right answer is to redirect the user to another page that doesn’t enable the user to click buy again. (Forgive me for using a Java-based example:
<http://www.theserverside.com/news/1365146/Redirect-After-Post>).

Unfortunately lots of web services **don’t use this technique**, they don’t do anything to solve the problem. Or they use JavaScript (which obviously doesn’t work if the user has JavaScript disabled).

But your game **should** do something. When the user initiates a transaction, change the state of the user-interface immediately, before the next event-loop, so that re-initiating the transaction is no longer possible.

What does your server do?

What could go wrong?



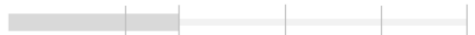
You should solve this problem for your users' sakes.

But this problem doesn't only happen for client-to-server transactions. What about server-to-server transactions...?

**"My account ... was billed
today for over 500 dollars in
15 dollar increments."**

-- Warhammer Online customer

What could go wrong?



Hmmm.... I guess not everyone handles this problem on the server side...

Idempotent transactions to the rescue

What could go wrong?



Well, there is a solution to this problem: idempotent transactions!

Idempotent transactions to the rescue *different from impotent

What could go wrong?



No... get your head out of the gutter 😊

IDEMPOTENT [ahy-duhm-poht-nt]
=> can be applied multiple times
without changing the result

What could go wrong?



Idempotent transactions can be run multiple times but even if they are the correct and desired result is always the outcome.

buy(item)

What could go wrong?



So here is a non-idempotent transaction.

buy(item, GUID) now with idempotency™

What could go wrong?



And here I've modified the transaction: now with idempotency. Incidentally I'm trademarking the term so that you have to pay me money each time you write any code that uses it. I'll include my checking account number later in the presentation – direct deposit means I don't have to cash all the checks you'll be sending; I can just order mai tais on the beach in perfect comfort, thanks!

So... what is this GUID thing? It stands for Globally Unique Identifier. What that means is that if each of you in the audience (several hundred folks) each run several thousand servers, each of which generates several thousand GUIDs per second, then, when the universe turns into a cold, dark ball of matter, none of should have generated identical GUIDs. Yeah, unique.

So the cool thing about this property is that we can assign GUIDs to each transaction to track them.

```
create table items
... item fields
transactId GUID UNIQUE
end
```

What could go wrong?



Here is some fakey-fake SQL code to build an item table. I've left out all the interesting fields your game uses, and only shown the GUID. It has a GUID field which includes a uniqueness constraint so that the same GUID can't be inserted into the table twice. Incidentally, this requires an index on the GUID column to make it work properly.

So the first time a user initiates a buy transaction, it should be successful. And the second time, it won't be because SQL won't let the same GUID be inserted into the table twice – SQL will trigger a uniqueness constraint violation, which we can report to the user. Great! But what we should really do is detect that error, and convert it back into a SUCCESS code for the user, because ultimately the transaction they initiated was completed correctly.

And we'll use this trick to solve other more complicated problems later.

Error: Invalid state transition

What could go wrong?



So let's tackle another problem!

Game server executes partial transaction

Game server talks to credit-card processor

Game server finishes transaction

What could go wrong?



Here's a pattern that programmers write that when they think that the application code they write is the primary driver of state, and that the database is "dumb storage", like a file-system. At another company I worked at, this pattern pervaded the billing system, and led to a lot of customer support problems.

First transaction: add something to the player's account.

Second transaction: go get payment.

Third transaction: fix up the player's account with the final details.

What's wrong with this?

Game server executes partial transaction

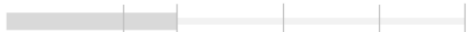
=> DB now in invalid state

Game server talks to credit-card processor

Game server finishes transaction

=> DB becomes valid again

What could go wrong?



What's wrong with this? The database contains results that are incorrect according to our defined business rules. And "briefly" here means from seconds to minutes, which is a long time in CPU-land. All sorts of failures can happen that prevent later transactions from completing, meaning that the database stays invalid.

Again, while this code **looks** like a straw-man, this pattern was repeated all through the billing system. And I expect that similar types of code exist through systems all over the world.

The fundamental problem is that databases are supposed to use transactions to maintain their validity across transactions.

May seem obvious: after every commit the DB must be in a valid state

What could go wrong?



After every transaction is completed on the database, SQL ensures that the database does not contain corrupt data, and that the results of the transactions are persisted. But what is important to us as developers is that the business rules which **we** define must be correct both before and after each transaction completes. SQL doesn't "know" what these rules are; it's our job to write code that ensures that the system is internally consistent.

This is the C in ACID

Atomicity - commit all or nothing

Consistency - data valid before and after

Isolation - intermediate data not visible

Durability - must persist after transaction

What could go wrong?



In SQL, this property is called “consistency”.

SQL does ACID

What could go wrong?



... 😊 ...

SQL does ACID

we need to ensure our data is meaningful

What could go wrong?



We need to maintain the consistency of our data according to our business rules by using the consistency property of SQL that ensures that are transactions are completed according to the rules we encode.

Error: Distributed transaction failure

What could go wrong?



So let's talk about a problem that more and more developers are getting to experience as games grow more popular. As we build online services with ever more users, it becomes more necessary to build distributed systems in order to scale to the hundreds of thousands or millions of users who play our games.

```
GameSrv_TradeItem (...) {  
    DB1->Send(p1, ADD, item);  
  
    DB2->Send(p2, REMOVE, item);  
}
```

What could go wrong?



Here's an item trade transaction. Because we have many users and can't afford a database big and fast enough for our user-population, we've "sharded" the data so that everyone sitting in the back of the room has their character records stored in database 1, and everyone in the front half (me included) has their character records stored in database 2. So when I give you an item, we've got to remove the item from my database and add it to your database – a distributed transaction.

So... what could go wrong?

```
GameSrv_TradeItem (...) {  
    DB1->Send(p1, ADD, item);  
    ... crash here ...  
    DB2->Send(p2, REMOVE, item);  
}
```

What could go wrong?

Ooops; that didn't go so well!

So what can we do to correct this problem?

Ignore the error

What could go wrong?



Well, the easiest solution is always best to consider first...

Ignore the error tech support will fix

What could go wrong?



At least we'll create lots of customer support jobs, which might help reduce the unemployment problem.

Ignore the error
tech support will fix
ask hackers not to exploit

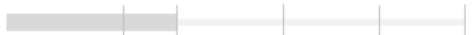
What could go wrong?



But we unfortunately have to ask hackers not to take advantage of the problem we've created; maybe we can ask them nicely.

Rollback the transaction

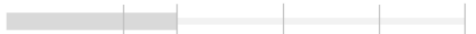
What could go wrong?



Perhaps we can roll back the transaction; SQL has that capability, right?

Rollback the transaction and hope rollback doesn't fail too

What could go wrong?



Rollback occurs infrequently, so the likelihood is that the rollback code path won't be well tested, and probably won't work when we need it most. *sigh*

Two phase commit

What could go wrong?



Or we can use two-phase commit solutions like MS-DTC (Microsoft Distributed Transaction Coordinator) or similar solutions for other platforms. Behind the scenes these solutions are using a form of manual transaction commit. They run all the transaction code across all the databases, but don't commit the results until all databases indicate that the transaction is going to succeed, then commit them "all at once".

Coordinator: ready one?

DB one: ready!

Coordinator: ready two?

DB two: ready!

Coordinator: okay.... Go!

Two phase commit

KILLS DB performance

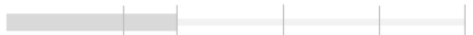
What could go wrong?



Unfortunately this solution causes a massive performance hit.

Two phase commit KILLS DB performance **NOT** guaranteed

What could go wrong?



And it doesn't always work. Check out the literature in two-phase commit and read about "in doubt" results. Effectively it means that while two-phase usually works, when it doesn't, it doesn't.

Solution:

Transaction queuing

What could go wrong?



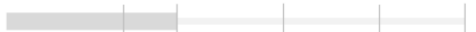
```
GameSrv_TradeItem (...) {  
    DB1->Send(p1, ADD, item);  
    ... crash here ...  
    DB2->Send(p2, REMOVE, item);  
}
```

What could go wrong?

Here's the code from before. Let's start by taking out the bad parts...

```
GameSrv_TradeItem (...) {  
    DB1->Send(p1, ADD, item);  
  
    DB2->Send(p2, REMOVE, item);  
}
```

What could go wrong?



Okay, much better. Now let's change this to use transaction queuing.

```
GameSrv_TradeItem (...) {
```

```
    DB2->Trade(p2, p1, DB1, item);
```

```
}
```

What could go wrong?



Well heck, it looks like all I did is move the problem somewhere else! But now another programmer can write that code and fix the problem so it's not on my plate anymore, and I get to go home early 😊

DB2:

remove(p2, item)

queue-add(DB1, p1, item)

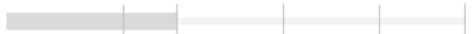
What could go wrong?

So let's talk about what that other programmer does.

I'm giving you an item, so the programmer writes the transaction so that the item is removed from my database. But the item isn't added to yours. Instead, let's write a "promise" to add it to yours "eventually".

DB2: **begin transaction**
remove(p2, item)
queue-add(DB1, p1, item)
commit transaction

What could go wrong?



And we'll wrap all that code in a transaction so that both steps occur together or not-at-all

DB2: begin transaction
remove(p2, item)
queue-add(DB1, p1, item)
commit transaction

What could go wrong?



So how do we implement that “promise” for our item trade?

```
worker: while true do  
  get-transaction (&src, &dst, &trans)  
  execute-transaction(dst, trans)  
  delete-transaction(src, trans)  
end
```

What could go wrong?



Well, we'll create a worker process on another server which monitors the database for promises, and makes sure they come true. Even if a database maintenance occurs, when the database (and worker) are restarted, the transaction will eventually complete. This is known as "eventual consistency".

What if worker keeps redoing the same work?

What could go wrong?



... But ... what happens if the worker keeps redoing that same transaction; won't we create lots of items?

What if worker keeps redoing the same work?

Make the work idempotent

What could go wrong?



Well, let's just use that idempotency trick we learned earlier (and remember to send me payment for using it).

Check out ZeroMQ for work-queuing

What could go wrong?



Transaction queuing is great stuff, and I could talk about it for hours. But instead I'll refer you to a great, free, open-source, well-documented solution to implement queuing behaviors (worker task-queues and such) that works across many languages: ZeroMQ.

... pause for breath ...

Whew! In my GDC presentation in 2010 on Security, “Developers vs. Cybercriminals: Protecting Your MMO from online crime”

(<http://www.slideshare.net/EnMasseEnt/developers-vs-cybercriminals-protecting-your-mmo-from-online-crime-3589535>), I packed quite a lot of material into a one-hour presentation, so much so that folks suggested I slow down a bit (actually, a lot). How am I doing this year?

Reliable Error Handling

So, since we're planning for failure, let's move on and talk about how to handle the inevitable errors that users are going to experience when playing our games.

Something bad happened... now what?!?

Something bad happened



When errors occur, what **should** we do? I know, let's display an error message to the user!



Something bad happened

Well, here's what developers should do: display a helpful error message.

Hmmm... maybe helpful isn't the right word. But most programmers aren't so good at writing error messages and communicating with, you know, actual soft-n-squishy human beings with those yucky emotions. That's why we went into programming in the first place: lack of social skills.

So when an error dialog like this pops up, what does the user do? Well, they'll call customer support, right?

Iceberg principle: only some customers will ask for help

Something bad happened



Unfortunately, not all users are going to call your tech support department. And I'm one of them! I hate calling support because most times the folks who work there, even if they're well-intentioned and not burned out (yet), can't solve the problem.

... the rest RAGE QUIT

Something bad happened



So many users (like me) will leave the game.

What does Customer Support do?

Something bad happened



But imagine your users contact customer support? What does a CS agent do?

The user complains about a problem, but from the error message it probably isn't even clear what the user was doing.

And with a two-hour wait queue, by the time the agent sends some questions back to the user, the user is already offline.

So the next day when the user replies, a new agent handles the problem. And s/he doesn't have an answer, because it's fundamentally a problem with the game code.

So after escalating through all three tiers of the traditional customer support department, what does a senior agent do? They call the Operations Team.

What does the Operations Team do?

Something bad happened



And what does the Ops Team do to solve the problem? Similarly, they escalate the problem too.

Call the devs

Something bad happened



And eventually call in the development team...

Call the devs after rebooting three times

Something bad happened



And eventually call in the development team... after implementing the usual escalation procedure.

STOP!

Something bad happened



STOP! This is all foolishness! What **should** we do?

Log the error

Something bad happened



Well, logging the error is a good start.

Log the error

Does anyone read logs?

Something bad happened



But does anyone read the error logs? See, in most companies the logs are all stored in one big file per day across multiple servers, and the information doesn't get aggregated in a useful way, so no developer goes to the effort of actually reading the log files, assuming they even have access to the logs.

Separate informational logs from error logs

Something bad happened



One possible solution: write different types of logging to different destinations so that the information can be summarized more effectively.

For Guild Wars we had three different error logs for each service:

Informational: stuff that users are doing: login-success, login-failure-bad-password-or-unknown-user, logout, add-friend, etc.

Error: stuff that went wrong that wasn't anticipated: cannot-access-the-dang-database-permission-denied-dammit

Debug: hmmm... other stuff

Then, at the end of each day, each of several hundred servers would send their error logs to the entire programming team. Needless to say, error conditions got fixed quickly because no one wanted the shame of their code spamming all of the programming team.

Tragically, the debugging logs ended up filled with cruft, and no one ever read them again.

Another solution is to use a tool like Splunk, which is like "Google for log files"; it completely kicks ass; check it out.

Bad error: Resource not found 404



Something bad happened



So, logging is good. What else can we do?

Let's work on giving the user a better error dialog. Well, most of our games are turning into web services; let's use HTTP error codes to provide more information.

This blows! The information conveyed in HTTP codes isn't enough by itself to provide enough diagnostic information

What about the user? And customer support?

Something bad happened



Provide enough information to diagnose or even ***fix*** the problem!

Better error:

Fancy message with [link](#)

34-15-3-743



Something bad happened

Here's a better solution:

1. Tell the user something meaningful about the error and suggest a solution *right there*.
2. Provide a link to an external site that you run that contains even more information. How about a wiki site or "answers" site like Stack Overflow, which is what we're doing for TERA (<http://tera.enmasse.com>), the MMO I'm presently working on? Then, not only can your support team provide possible solutions to the problem, but so can your users!
3. Provide a proper error code.

34-15-3-743

Wait, what?!?

Something bad happened



Huh?

Error 34 – routing error
Service 15 – cache server
Module 3 – forwarder.cpp
Line 743 – `__LINE__`

Something bad happened



Provide an error code that immediately identifies the source of the problem, which can include a lot of information helpful to programmers. This type of error code has additional useful properties.

1. When launching in multiple languages, the CS team can simply report the error code from a foreign language user instead of translating the error string back into English (or whatever language the devs speak).
2. Users can also search for this error code using Google, and find alternate solutions that might not be included on your support site.

Which do you think is easier to diagnose? HTTP 500? Or this? Give your users and your CS team a fighting chance at solving the problem!

Good error messages lead to faster fixes

Something bad happened



Security!

Okay, now for a change of pace.

You've made a successful game; users want to play it and perhaps even pay you.
Unfortunately, it's now a compelling target for the bad guys!

The **bad** guys:

- * professional cybercriminals
- * lots of resources
- * lots of stolen accounts for testing
- * they read security literature

Stopping the bad guys



There are lots of folks to worry about: script-kiddies, who want to take down your servers through denial-of-service attacks; griefers, who want your users to suffer; casual hackers, who just want eke out a better play experience in your game, and more.

In fact professional hackers are going to be your biggest problem. They get paid to hack our games. There is a lot of money in the hacking business, so the hackers are probably paid more money than we are, which also makes them smarter than us.

They also have the opportunity to research what we do and publish. They might even be here in the audience. Anyone want to own up to being a professional account-stealer? No? Well, this might not be the best venue to come clean...

Top vulnerability: Injection attacks

Stopping the bad guys



Each year several security organizations post lists of security vulnerabilities from the past 12 months. One such organization is OWASP. And each year the list looks pretty much the same. So while it might seem elementary to cover something like SQL injection attacks, given how commonly the problem is exploited in recent successful attacks against Sony, HB Gary, Sony, Eidos, Sony, RockYou, Sony and others (like Sony), I'm going to talk about the problem in detail.

Some typical PHP code

```
$sql = "select * from Users where Name = '" + $name + "'"
$query = $db->prepare($sql);
$query->execute();
```

Stopping the bad guys



Here is some typical PHP code, which is to say awful code since PHP takes the "worse is better" philosophy to such an extreme. PHP is a set of security vulnerabilities packaged as a programming language. But since it is the most popular web language in the world, here's an example of some common code that talks to a database to get information about a user.

Some typical PHP code

```
$sql = "select * from Users where Name = '" + $name + "'"
```

```
$query = $db->prepare($sql);
```

```
$query->execute();
```

what happens when \$name is
' or 1=1 --

Stopping the bad guys



But what happens when a hacker sends an improper name to the web service on login?

Your query becomes:

```
select *  
from Users  
where Name = " or 1=1
```

Stopping the bad guys



Well, this query would select **all** fields for **all** users from the database. Gulp!

Solutions

- * Stored procedures
- * String escaping
- * Parameterization

Stopping the bad guys



Stored procedures

Stopping the bad guys



With stored procedures you're forced to pass “bound” parameters to the procedure instead of composing SQL queries using string concatenation, which is the source of the problem with SQL injection errors.

Vulnerable stored procedure

```
CREATE PROC BadProc (@param varchar(256)) as  
  DECLARE @ cmd varchar(1024)  
  SET @cmd = 'select * from foo where bar = ' + @param  
  EXECUTE(@cmd)
```

<<< **ARRRGH!** >>>

Stopping the bad guys



Of course, it's still possible to write a stored procedure that is vulnerable to injection, as shown above.

If you *have* to write dynamic SQL, check out http://www.sommarskog.se/dynamic_sql.html; the author is wicked smart.

SQL Escaping

'bob' -----> 'bob'

" or 1=1--' -----> "" or 1=1--'

Stopping the bad guys



Probably the most commonly adopted solution is to “escape” the SQL string. Just like you can create special escape sequences in strings – \n for newline, \t for tab, and the like – it’s also possible to escape quote characters so they can’t be exploited.

SQL Escaping

'bob' -----> 'bob'

" or 1=1--' -----> " or 1=1--'

Stopping the bad guys



Here is the previous slide recolored to highlight the escaped quote, which finesses the injection attack.

The problem with escaping is that, in many programs, there can be many layers of code responsible for performing a transaction. Which part of the code is responsible for escaping? High-level? Mid-level? Low-level? When many programmers are working on a project it can be easy to lose track of which person is responsible for getting this right. It's a brittle solution to the problem.

There is a better way...

Parameterization

Stopping the bad guys



Yup. Better.

Some typical (but no longer truly awful) PHP code

```
$sql = "select * from Users where Name = :name"
```

```
$query = $db->prepare($sql);
```

```
$query->execute( array(':name' => $name) );
```

-- using the **PHP PDO** library

Stopping the bad guys



Here's that same PHP code, fixed up to use parameterization. Instead of composing a SQL query string "on the fly", we instead separate the parameters from the query, and provide named parameters to the query execution function. Simple, huh?

And parameterization will make your code faster

Stopping the bad guys



Parameterized queries get cached by SQL as a compiled execution plan (including expensive security verification checks). This can make your SQL queries hundreds or thousands of times faster. Remember that billing system I was talking about earlier? A new, very expensive server didn't do spit to make the database run faster, but using SQL query parameterization basically solved the scaling problems the billing team was having.

Securing your network protocol

Network protocol requirements:

- * Encryption
- * Validation
- * Rate-limiting

Stopping the bad guys



Protocol Encryption

Stopping the bad guys



Why encrypt the network packet stream? It's not, as some folks might think, to protect your servers from being reverse-engineered, or to make them harder to hack. Hackers already have the client-side code on their computers, so they've already got everything they need. Instead, encryption is to protect your **users**, who will be sending Personally Identifiable Information (PII) like email addresses, names, credit-card information, and other stuff they'd prefer the world not to see. Your job is to create trust among your players, so don't screw them by sending their personal stuff over an unsecured connection.

Writing your own "encryption" algorithm is not encryption

Stopping the bad guys



The first mistake many folks make in encryption is writing their own encryption solution.

Wifi communications are totally insecure because some of the best minds in the hardware industry messed up the encryption protocols. WEP, WPA, WPA2? All easily crackable.

**"Anyone can invent an encryption algorithm they themselves can't break; it's much harder to invent one that no one else can break".
-- Bruce Schneier**

Stopping the bad guys 

Bruce Schneier -- **the** cryptography guru -- tells it like it is.

So... go use an algorithm written by really smart folks that has been peer-reviewed by really smart folks.

AES256 is a good one. But don't write it yourself; use a library that was written by really smart folks and has withstood the test of time so you can avoid all the common implementation mistakes that **they** made in their first version.

Using a symmetric key embedded in the client is not encryption

Stopping the bad guys



Another common mistake: if you're using "symmetric key encryption" (AES256, RC4), the security of the key is important. If you embed the symmetric key in your application (or worse, send it through the network unencrypted), all hackers have to do is look for it. And remember – they're smart!

Share encryption keys with Diffie-Hellman key exchange protocol

Stopping the bad guys



... because it works.

Crypto requirements

- * Do not write your own crypto method
- * Use well-understood algorithms
- * Do not store keys with application
- * Read the security literature

Stopping the bad guys



<http://security.stackexchange.com/questions/2202/lessons-learned-and-misconceptions-regarding-encryption-and-cryptology>

Protocol validation

Stopping the bad guys



Protocol validation is all about ensuring that whatever message you receive is both syntactically valid (well formed, like a grammatically correct sentence), and semantically valid (meaningful, not grammatically correct but nonsense).

How about this?

```
int recv_msg(char * data, unsigned bytes) {  
    if (bytes < sizeof(Header)) return false;  
    Header * hdr = (Header *) data; bytes -= sizeof(hdr);  
    char *base = data;  
    char * str1= data; data += strlen_s(data, bytes-(data-base))+1;  
    char * str2= data; data += strlen_s(data, bytes-(data-base))+1;  
    char * str3 = hdr->someflag ? data : "";  
    ... etc.
```

Stopping the bad guys



Here's some code that's similar to code I (and many other programmers) wrote to validate binary messages passed between game services. A fixed-size binary header, some variable-length strings, some optional.

(Oops)

```
int recv_msg (char * data, unsigned bytes) {  
    if (bytes < sizeof(Header)) return false;  
    Header * hdr = (Header *) data; bytes -= sizeof(*hdr);  
    char *base = data;  
    char * str1= data; data += strlen_s(data, bytes-(data-base))+1;  
    char * str2= data; data += strlen_s(data, bytes-(data-base))+1;  
    char * str3 = hdr->someflag ? data : "";  
    ... etc.
```

Stopping the bad guys



Crikey; it doesn't work very well, does it? Don't do this. Even if you do this, you're going to shoot yourself later when you want to send (for example) UTF8 strings, UCS2 strings, floats, arrays, variable-length arrays, arrays of structures, floating point values, and the like.

Use:

MsgPack

Protocol buffers

Thrift

XML / JSON (if you must)

Stopping the bad guys



Don't reinvent the wheel; either use some of the many solutions out there that have already been proven to work, or at least read their code and write a similar data-driven solution.

But also remember that you'll need to validate the meaning of the parameters – is negative infinity valid in the context of your game? What about NaN (not a number), infinity, or negative zero?

Service rate-limiting

Stopping the bad guys



Once you've ensured that your server will only accept valid messages, you still have to worry about what happens when you get messages more frequently than they should arrive.

Many games have problems with "speed-hacking", where a game client that should only send five messages a second sends fifty, and consequently the sender has a 10x attack advantage over other players. Oops.

Or another example: about ten months ago (roughly a year before the commercial launch of our game), we saw over one million login attempts in a single day from a single IP address. These attacks were an attempt to crack accounts on our system. The pattern of the attack was that it tried many different email addresses, each with a single password. That is, not a brute-force attack to guess the password for a single account, but a single login attempt for each of one million accounts. What gives? My suspicion is that the attacker had a database of stolen accounts with their passwords, and was just trying to determine whether (a) those users had accounts on our system and (b) if the account owners were using the same password on both systems.

If you don't have rate-limiting, this kind of attack can be a problem for two reasons: (1) players who re-use the same password (probably most of them) are vulnerable to account-hacking and (2) your game can get knocked offline due to the hacking attempts.

Rate-limiting transactions prevents hackers from being able to run millions of attacks against your service from a single IP address. But the bad guys have lots of computers – they can rent out bot-nets – so we need to craft a solution that prevents even distributed attacks.

```
ErrorCode CSocket::OnPlayerLogin (const Msg & m) {  
    if (!m_rateLimiter.AddTime(  
        LOGIN_COST_MILLISECONDS, // 20*1000  
        MAX_LOGIN_COST_MILLISECONDS)) // 20*1000*10  
        return ERROR_LOGIN_RATE_LIMIT;  
    ... actual login code here ...  
}
```

Stopping the bad guys 

Here's how rate-limiting works for login-transactions.

I wrote a blog article on the subject here: <http://www.codeofhonor.com/blog/using-transaction-rate-limiting-to-improve-service-reliability>

It's a good idea to rate-limit even on internal services; when working on billing integration for Guild Wars I discovered that one of the external billing teams "attacked" our service through our API – they submitted millions of transactions in a short time-window. So it turns out to be necessary to perform rate-limiting even on internal interfaces to provide maximum reliability to users.

```
bool AddTime (int costMs, int maxCostMs) {  
    int currTimeMs = (int) GetTickCount();  
    if (currTimeMs - m_timeMs > 0)  
        m_timeMs = currTimeMs;  
    int newTimeMs = m_timeMs + costMs;  
    if (newTimeMs - currTimeMs >= maxCostMs)  
        return false;  
    m_timeMs = newTimeMs;  
    return true;  
} // thx GlenK for bug fix
```

Stopping the bad guys



You can see from looking at this sample how tough it is to talk about code using PowerPoint!

See <http://www.codeofhonor.com/blog/using-transaction-rate-limiting-to-improve-service-reliability>

And thanks GlenK for fixing a bug in ~2009 that I wrote in ~2001.

Password storage?!?

Finally, I want to talk about password storage. Remember when I said “plan for failure” ? Well, plan for your account database getting ripped-off by bad guys. It happens a lot more frequently than anyone wants to talk about. And when those database get ripped off (Sony – ~30 million accounts, RockYou -- ~30 million accounts) along with the passwords, it makes it easier for the bad guys to steal accounts on other services because users recycle the same passwords across all systems. C’mon, like you’ve never done it...

Many developers have used solutions like MD5-hashed passwords, or SHA1-hashed, or even those hash algorithms with “salt”. But they’re pretty easy to crack for bad-guys who have lots of resources (like professional game-gold sellers).

Coda Hale says “These are all *general purpose* hash functions, designed to calculate a digest of huge amounts of data in as short a time as possible.”
(<http://codahale.com/how-to-safely-store-a-password/>)

So don’t use ‘em.

Use bcrypt

Stopping the bad guys



<http://codahale.com/how-to-safely-store-a-password/>

Conclusion

Security is a continuous process; you are never done

Game over man, game over



Security is asymmetric warfare: we have to create a secure network that's resistant to attack all the time on all fronts. Hackers only have to break in one time at one weak point to win.

On the plus side, for those of you considering careers in security, that means you'll never lack for employment; it's a growth field!

Increase player retention by creating robust software

Game over man, game over



And to sum up the point of this whole talk, the idea is that by creating great games built on top of secure, reliable, robust online services, we're more likely to end up with lots of players, which is a win.

Thanks to
Matwood
Aaron LeMay
Aria Brickner-McDonald

Game over man, game over



Questions?

Resources:

Scalability - <http://highscalability.com/>

Queuing - <http://www.zeromq.org/>

Parameterization - <http://php.net/manual/en/pdo.prepare.php>

Dynamic stored-procedure queries - http://www.sommarskog.se/dynamic_sql.html

Service rate-limiting - <http://www.codeofhonor.com/blog/using-transaction-rate-limiting-to-improve-service-reliability>

Storing passwords with bcrypt - <http://codahale.com/how-to-safely-store-a-password/>

Diffie-Hellman cryptographic key exchange - http://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

AES encryption implementations - http://en.wikipedia.org/wiki/AES_implementations

“If I have seen further it is by standing on ye shoulders of Giants.”

-- Isaac Newton

Here are links to some of those folks, with my blog linked in there to give you the idea that I might be one of ‘em ☺

Thanks for making it all the way to the end

Mail me at “pat” at “codeofhonor.com”