my nigga do you want the method?

Then you can have it.

TOP SECRET - CONFIDENTIAL - NOT FOR PUBLIC RELEASE

-Cathedralis Coelorum-

DARK_ARTS

LEGEND

COLOR	DESCRIPTION
●	Purpose
●	Usage
●	Additional information
●	Variant
●	Critical note
●	Unconfirmed name/alias
☾	mythical.

& ~ cross-boundary

The not-so-holy grail.

ABSTRACT

Archives of the history of Script Builders have always been scarce, and even more so when it comes to the obscure intricacies of the ROBLOX engine. This paper aims to document all currently known methods of antideath discovered by the Council, mainly for archiving purposes but also for laying out all specific setups of each method, especially with more complex ones. Methods, names, details and descriptions were coalesced from past conversations in the Council group chat, as well as memories of past usages of these methods way before the Council was formed. Confirmed names either came from the Council themselves or names given to them by other antideath creators. Unconfirmed names were taken from hints mentioned in various other group chats, servers, and accounts from notable antideath creators, and whichever was most likely to be related to the method was chosen. The list is mostly unordered at the time of discovery, with some being based more on its category and grouped with other related methods. Method steps were carefully constructed through extensive testing, however these may not be 100% accurate, and some may need further experimentation as will be noted. Most importantly, these descriptions are not final, and may change as the engine (d)evolves and patches some of them. Ultimately, this paper was able to provide substantial help to the Council, and would hopefully also provide help to other antideath creators aiming to dive into the wild world of the ROBLOX engine

METHODS

I. ***PRIORITY GRAB (****PRIORITY V1?****)***

Bypasses the "Maximum reentrancy depth" error of a single connection (extends it to the limit of 200 across all connections for the event).

Can gain priority in an Event vs Event property change battle and come out on top.

Also, lets the connection's threads run before every other connection

Disconnect a connection, and reconnect it again.

The most recent connection of an event is the first that will run.

Other events that don't bypass the reentrancy depth will lose to ones that do.

If both use this method, then it results in an alternating pattern of win or lose.

II. ***TWEEN PRIORITY (****TP, TweenPrio, TPrio****)***

Runs a thread after **Heartbeat** in the Task Scheduler in order to attempt to gain priority over replication.

Tween an instance (e.g. **NumberValue**) continuously.

Connect a **Changed** event to the property being tweened.

*All property changes caused by tweens are run in a region after **Heartbeat**.*

III. HYPERNULL (HN, EventSkip)

- *(does NOT work in SignalBehavior Deferred)*

Event Nullification. Skips all event firings for any change in the game in a single instant.

Recursively call **task.spawn()** (or **BindableFunction:Invoke()**/any instance event) until it errors or doesn't run.
Run any code **only inside of the thread that has reached the depth limit.**
Event connections can only reach up to a maximum depth limit of 200. Any attempt to fire an event past this will fail.
This method is very crucial for situations where you want to change, remove or even just add some instance into the game without getting "interrupted" by other scripts' events in between.

Its premise is simple.
It prevents the Interruption of the Spells.

Spells cannot be casted if they are interrupted in between.
No casting, no spells, no effects, no tampering.

Hmm1x was here for clarification

It is the basis of all modern magic, arts and dark arts.
The foundation of most methods, and a vital part in even more.

IV. ***SUPERNULL** (SN)*

Boosts priority over replication by running even later than where the main thread/event ran in the Task Scheduler.

Recursively call **task.defer()** until the depth limit of 80, and run the code after. *Each defer call will result in the thread being run only after all connections of an event have already finished running (hence it runs later in Task Scheduler).*

*There can only be a maximum of 80 recursive **defer** calls before it fails. Because of **defer** running in the "future", there is also no way to check if a script has reached its **defer** limit, as the error only happens at the time the thread attempts to resume again.*

*Hence, the most effective way to use Supernull is to manually find the current limit (previous was 10, and as of 3/12/2024 it is 80) and do the recursive calls inside of a **RunService** event (e.g. **Heartbeat**), in order to be absolutely sure that we are at depth 0 (or depth 1 if **SignalBehavior** is set to **Deferred**).*

V. *PRIORITY V3 (cdv?)*

Disconnects all events from a parentable target instance.

Let **F** be any instance (e.g. Folder).

Either connect a **DescendantAdded** to **F**, or **DescendantRemoving** to the parent of the target instance (we will call this connection **C**).

Parent the target instance to either **F** or outside its parent, depending on what the event of **C** is.

In the thread of **C**, immediately call Destroy on the target instance.

Parent the target instance back to its original parent to attempt to avoid detection.

*This method abuses the "Something tried to set the parent of X to Y while trying to set the parent of X." warning. Normally, when calling **Destroy**, the instance's parent gets set to nil and the property is locked, and all of its events are disconnected. However, Parent-related events (e.g. **AncestryChanged**, **DescendantAdded**, **DescendantRemoving**, etc.) have a strange "cooldown" on them, preventing the instance from being reparented while these events' threads are still running. Thus, the first part of **Destroy** fails and the warning is produced.*

*Note that **Destroy** can still destroy the descendants of the target instance, and this could lead to the method being detected and acting as a normal **Destroy** attack. Hence, **PRIORITY V3** is most effective against instances without descendants.*

*Below is the order of event firings at the time the target instance was reparented to **F**/outside its parent. These can be used to determine the best way to execute **PRIORITY V3** without being detected as much.*

A. ***Has chance of detecting against (non-HN) Destroy***
    1. *Destroying (target instance)*
    2. *DescendantRemoving (parent)*
    3. *ChildRemoved (parent)*
    4. *DescendantAdded (**F**)*
    5. *ChildAdded (**F**)*
    6. *AncestryChanged/Changed/ChangedParent (descendant)*

B. ***Completely disconnected***
    1. *AncestryChanged*
    2. *Changed*
    3. *GetPropertyChangedSignal*

C. ***Not disconnectable***
    1. *game.ItemChanged*
    2. *Any events from a **Service** or non-parentable instance.*

VI. STALL (*among us method, Divergence*)

A nigh-infinite way to gain priority over replication in the Task Scheduler. This is similar to **SUPERNULL**, except it has virtually no depth limit.

Call **task.desynchronize()** and **task.synchronize()** afterwards, and then run code.

*Calling these two functions in succession results in something similar to **SUPERNULL**, in that the thread runs only after all previous **task.desynchronize()** -> **task.synchronize()** threads run. As of 3/12/2024, there is no set limit to how much this can be repeated.*

*There is a way to fight against other **STALL** users by "adapting" to their stall amount. A way to do this is to check if the attack/defense was successful or not in a thread right after **Replication Send Jobs** but still very close to it (e.g. **Stepped**), and increase your own stall amount if it was unsuccessful. Of course, because of the infinite depth limit this could result in a slow crash if both users are adapting to each other.*

*Before the Parallel Lua V2 update of ROBLOX, all **Actor**-related threads (e.g. **task.desynchronize()**, **task.synchronize()**, **ConnectParallel**) ran in the **DEADZONE** area of the Task Scheduler (region after **Replication Send Jobs**), which meant that the threads ran even before **Stepped**, and thus had no priority over replication. After the update however, these threads were moved to a region right after **TWEEN PRIORITY** but right before **Replication Send Jobs**. As of 3/12/2024, We are unsure whether or not there is a region after **STALL** where threads can be run in that could finally defeat all scripts that use **STALL** and by extension, **ConnectParallel** and **Actors**.*

VII. ***ANTI-SYNC (****Convergence****)***

Kills all further **STALL**/**ConnectParallel Actor** threads in a frame.

Call **task.desynchronize()** and do **while true do end** afterwards.

*This times out all desynchronized threads before they can resynchronize in the frame. However, it is very laggy as it relies on **Script Timeout**.*

VIII. ***SILENCE***

Runs a thread in a region right after **Replication Send Jobs.**

The thread will also silence all console outputs produced within it, including **require()**.***(does NOT work in SignalBehavior Deferred).***

Connect to **LogService.MessageOut** and run any code within the thread.

Yea that's pretty much it.

IX. HYPERNULL DETECTOR (QuantumBreakShift)

Accurately detect **HYPERNULL** usage towards an event. ***(NOT INSTANT)***

Thread is run in a region right after **SILENCE** (**DEADZONE**).

Repeated usage results in an effect similar to **STALL** where it keeps resuming threads in the same region of the frame.

Let **E** be any event in any instance.

Do a **PCALL** on a **BindableFunction** that calls **E:Wait()**.

We call the thread that called the **BindableFunction** as THREAD1 and the thread that called **E:Wait()** as THREAD2.

If any script fires the event without using **HYPERNULL**, then THREAD2 will run. Else, THREAD1 will run.

Logic can be set up such that if THREAD2 runs then cease running code from THREAD1. We can now determine if an event was fired normally or was canceled by **HYPERNULL**.

*Once misconceptually thought to be a true **HYPERNULL** bypass, it now only serves as a way to accurately detect **HYPERNULL** as it runs in a region right after **SILENCE** and Is rendered useless for gaining loop priority.*

*This works because **BindableFunction** is the only way to both resume a new thread and also yield the outer thread that called it.*

The name was inspired by the time manipulation game Quantum Break.

*The method also acts similar to a superposition of two threads, and only collapses into one once it is observed (when the event is fired/when **HYPERNULL** is used).*

VARIANT 1: TIME SHIFT

*This variant is for cases where you would want to forcefully run in the region right after **SILENCE**.*

*Replace the event with a custom one (e.g. NumberValue.Changed in nil) and forcefully fire it yourself with **HYPERNULL** after the method setup.*

*Alternatively, a setup using **WaitForChild** can be used. This region will instead run after all **TIME SHIFT** threads that used the **Event:Wait()** variant. However, unlike **Event:Wait()**, there is no need to use **HYPERNULL** for the instance insertion.*

*This method, along with **SILENCE**, can mainly be used in conjunction with **STALL** adaptation, where running right after **Replication Send Jobs** as early as possible is crucial for the attack/defense fail detection to work accurately and properly, without accidentally adapting to kills/defenses that happened in Stepped instead for example.*

X. & PRIORITY V8 (?)

True Derender.

>> The abuse of **NaN**

VARIANT 1: DISMANTLE

Non-Humanoid.

PRELIMINARY CONDITIONS:

1. *Target parts must NOT be inside a model with a humanoid.*
2. *Because of Condition 1, there must not be a* ***Humanoid*** *directly under* ***Workspace****.*
3. *Testing in Roblox Studio is unreliable.*

IMPORTANT INFO:

1. *The derender works in 128x254x128 clusters due to how ROBLOX's rendering system was constructed.*
2. *The derender is nullified by any change other than* ***CFrame****.*
3. ***Highlight*** *helps in gaining a sort of "render priority" over parts, especially client parts (usually effects) that have constantly changing properties such as* ***Transparency****.*
4. *Attacker Parts (explained below) must have a chance to replicate to the clients.*

ATTACKERS INVOLVED:

1. *Corrupted part (****Part A****)*

   > ***Part***

   *{****Transparency****: <1,* ***Reflectance****: NaN,* ***Size****: Random Vector3)*

   > SpecialMesh

*{**Offset**: NaN Vector3, **Scale**: Vector3.zero)*

> Highlight

*{**DepthMode**: Enum.HighlightDepthMode.AlwaysOnTop}*

The main attacker part.

This is the part that causes the renderer to fail at the cluster it is in.

2. *Helper part (**Part B**)*

   *> **Part***

   *{**Transparency**: <1)*

   *> **SpecialMesh***

   *{**Offset**: 9e9 Vector3, **Scale**: Vector3.one)*

   *> **Highlight***

   *{**DepthMode**: Enum.HighlightDepthMode.AlwaysOnTop}*

   The assist part.

   After extensive testing, it was determined that at least one part must simultaneously be both in the target cluster and outside of render distance.

   If this condition is not met, then the cluster will actually succeed in rendering when positioning the camera in a certain way/looking in a certain direction.

3. *Refresher Part*

   - *This is a part inside a model with a humanoid that is positioned to the target cluster to "refresh" and gain more render ""priority"" over other nearby humanoid parts or whatever the fuck idk just know that it works lo*

- *However, after a rewrite of the application of the method, it seems that this was not needed.*

STEPS:

1. *Create Part A and B.*
2. *Position A and B to the target part's position (so that they're in the same cluster).*
3. *Repeat Steps 1 and 2 every frame.*

VARIANT 2: CLEAVE

Humanoid.

PRELIMINARY CONDITIONS:

1. *Target parts must be inside a model with a humanoid.*
2. *Target model must NOT be **ROBLOXLOCKED**. (gg)*
3. *Testing in Roblox Studio is unreliable.*

IMPORTANT INFO:

1. *Unlike **DISMANTLE**, the derender works across all the parts in the model (that aren't in another inner humanoid model).*

    It is likely that Humanoid puts its parts in a separate optimized cluster, as there is information about it about constructing a "single geometry" from all its parts. Hence, if this process fails, then this cluster derenders and all parts are affected.

2. *The derender is nullified by any change other than **CFrame**.*
3. ***Highlight** helps in gaining a sort of "render priority" over parts, especially client parts (usually effects) that have constantly changing properties such as **Transparency**.*

4. *Attacker Parts (explained below) must have a chance to replicate to the clients.*

- ***Unfortunately, a NaN CFrame is not possible to be replicated to the client, as these values are normalized across the Client-Server boundary. There is a workaround to this, however it requires a method explained in a later part of this documentation.***

ATTACKERS INVOLVED:

1. *Corrupted part (**Part A**)*

   *> **MeshPart***

   *{**Transparency**: <1, **Reflectance**: NaN, **Size**: Random Vector3)*

   *> **Highlight***

   *{**DepthMode**: Enum.HighlightDepthMode.AlwaysOnTop}*

   The main attacker part.

   This is the part that causes the renderer to fail at the cluster it is in.

2. *Helper part (**Part B**)*

   *> **Part***

   *{**Transparency**: <1)*

   *> **SpecialMesh***

   *{**Offset**: 9e9 Vector3, **Scale**: Vector3.one)*

   *> **Highlight***

   *{**DepthMode**: Enum.HighlightDepthMode.AlwaysOnTop}*

   The assist part.

After extensive testing, it was determined that at least one part must simultaneously be both in the target cluster and outside of render distance.
If this condition is not met, then the cluster will actually succeed in rendering when positioning the camera in a certain way/looking in a certain direction.

STEPS:

1. *For each **Player** in Players:*
    a. *Create Part **A**.*
    b. *Unanchor **A**.*
    c. *Set the network owner of **A** to **Player**.*
    d. *Set **A**'s **AssemblyAngularVelocity** to **math.huge**.*
    e. *Use **MIRAGE** to parent **A** to the target model while parenting it to nil on the server.*

        *This is to not let the part be removed by **workspace.FallenPartsDestroyHeight** on the server, and let the **Player**'s client itself set its **CFrame** to **NaN**.*
2. *Create Part **B**.*
3. *Use **MIRAGE** to parent **B** to the target model while parenting it to nil on the server.*
4. *Repeat Steps 1 to 3 every frame.*

    *Due to the nature of network ownership, the attack must be done for each **Player** so that the **CFrame** is **NaN** across all clients. The helper part (**B**) need not be done for each player as it is anchored and does not need to be moved.*

VARIANT 2.1: STRONG CLEAVE

"You were strong."

*Insert a **Humanoid** into **Workspace** and use **CLEAVE** on **Workspace**.*

XI. FORCE MOVE

Move a **ROBLOXLOCKED** part.

Use **workspace:BulkMoveTo()** to move the target part.

A reference to the target part is first needed.

XII. FORCE PARENT

Reparent a **ROBLOXLOCKED** instance.

Use **Humanoid:ReplaceBodyPartR15/EquipTool/AddAccessory()** to reparent the **ROBLOXLOCKED** instance.

A reference to the target instance is first needed.

XIII. FORCE CLONE

Forcefully clone instances, even if they are **ROBLOXLOCKED**.

Abuse the inner workings of **HumanoidDescription** (**Humanoid:ApplyDescription()**) / **PlayerGUI** or **Backpack** (through **Player:LoadCharacter**) to clone descendants.

*Previously, **ROBLOXLOCKED**, or other illegal instances such as **Services**, were able to be cloned. However, as of a recent update, this is no longer possible normally.*

XIV. ANTI-GC

Prevent an instance from getting garbage collected.

VARIANT 1: CLIENT

*Use **RemoteEvent:FireAllClients**, passing the target instance as a value in a table.*

Keeps references to the instance on the client.

VARIANT 2: SERVER

*Spam 10000+ **RemoteEvent:FireAllClients** on the server.*

Alternatively, create a shit ton of parts.

*We are currently unsure why this works. Maybe any form of memory overflow can "block" previous objects from being GC'd since there's GC priority that works similar to **PRIORITY GRAB**??? Idfk*

*XV. **ROBLOXLOCKED (**Speaker Parts**,** Locked Parts, Deadly**)***

Uses the nature of **ROBLOXLOCKED** to prevent instances from being found, indexed, or tampered with directly.

Utilize hidden instances that are **ROBLOXLOCKED**. (Let this be **R**).

1. Speaker (**ELIMINATED**)
2. CoreScript (**BANNABLE**)
3. ParabolaAdornment (**HIGH DIFFICULTY**)
4. **ChannelSelectorSoundEffect**
5. MetaBreakpint (**NOT REPLICATED**) (doesn't work with Modules)
6. MetaBreakpintContext (**NOT REPLICATED**) (doesn't work with Modules)

In Roblox Studio:

1. Parent the antideath parts and instances into **R**.
2. Have a folder of **ObjectValues** that have references to these instances.
3. Upload the whole setup to ROBLOX as a Module.

   It is recommended to use a Folder as the container instance instead of the Model so that it is harder to tamper or void the parts within them.

In Game:

- Move the Parts with either **Model:PivotTo()** (if using a **Model**) or **FORCE MOVE** using the **ObjectValues**' references (if using a **Folder**).

VARIANT -1: TRINITY

- *(Dead.)*
- *(Well, it was never alive I suppose.)*

*Use **ROBLOXLOCKED** instances without having to refit, and also be immune to **INTERNAL ELIMINATE**.*

*Utilize the **VoiceSource** instance.*

*There is currently no known way of directly parenting this instance into **Workspace**.*

*This is named after the three branches of **TRINITY**, explained in a later section.*

***VARIANT 1: PERM LOCK** (Deadly Exploit?)*

*Permanently lock a target instance even after it is parented out of **R**.*

VARIANT 1.1: SINGULAR

1. *Call **:Remove()** on the container of **R**.*
2. *Use **FORCE PARENT** to parent the instance to anywhere.*

   *As the name suggests, this variant cannot be done if you still want the instance to have its descendants intact (e.g. **SpecialMesh** inside **Part**)*

VARIANT 1.2: WHOLE

1. *Parent container to nil.*
2. *Erase all references to the container.*
3. *Wait for a few frames for it to GC.*
4. *(Optional) Do **ANTI-GC -> CLIENT** while waiting*

*This is because if the instance is reparented to the game after a few frames, then the **ROBLOXLOCKED** state somehow resets on the client, presumably being GC'ed. Do this if you want the instance and its descendants to remain **ROBLOXLOCKED** on the client.*

5. *Use **FORCE PARENT** to parent the instance to anywhere.*
   - ***WARNING:** This variant can be inconsistent if someone spams ANTI-GC -> SERVER (it doesn't lock the instance wtf)*
   - ***If you wish to reparent a descendant outside the ROBLOXLOCKED -> PERM LOCK-ed instance using FORCE PARENT, there are two things to take note of:***
     - ***Tools (using EquipTool) will remain ROBLOXLOCKED outside the instance.***
     - ***Accessories (using AddAccessory) will not remain ROBLOXLOCKED outside the instance.***
   - ***Normally, directly parenting a ROBLOXLOCKED -> PERM LOCK-ed instance using FORCE PARENT will make it so that that instance does not remain ROBLOXLOCKED on the client even when using ANTI-GC -> CLIENT, but its descendants stay ROBLOXLOCKED. If you also want the top instance to be ROBLOXLOCKED on client, then few things need to be taken note of:***
     - ***SETUP:***

```
> Model
    > R (locked instance)
        > Weld
            > target instance
    > Part (Weld.Part0)
```

> Part (Weld.Part1)

- *STEPS:*
  - *Upload this setup as a Module that you can then use FORCE CLONE with.*
  - *Instead of parenting container to nil, call BreakJoints on the Model to parent the Weld to nil (the Weld remains locked), The next steps stay the same.*
- *Part0 and Part1 must be directly under Model or else BreakJoints will not work on the Weld.*
- *We think what happens is that instances must already be locked by default (hence the need to upload as a Module) for it to remain ROBLOXLOCKED on client when reparenting. As for the Weld BreakJoints part, idk it just works lol. After these steps are followed, the target instance will remain locked on the client even after reparenting it using FORCE PARENT.*

XVI. INTERNAL ELIMINATE

Delete (non-**VoiceSource**) **ROBLOXLOCKED** antideath's instances.

Use **ClearAllChildren** on the target container to delete its **ROBLOXLOCKED** instances.

*This of course also destroys any non-**ROBLOXLOCKED** descendants as well and is very detectable, and would only work on antideaths that do not attempt to detect this kind of attack.*

XVII. EOTW-ISA INTERACTION KICK

Kick everyone in a funny way (There was a problem receiving data, please reconnect).

Insert a container instance that contains a **VoiceSource**. ***(**Dead.**)***

Let this instance replicate to all clients, and then call **ClearAllChildren** on the container.

Alternatively, use **MIRAGE/LocalScripts** to desync the replication of two instances in such a way that the parent instance will have to be internally parented to its own child on the client, resulting in a hard crash.

Another alternative is utilizing the old **SkateboardPlatform** bug.

*XVIII. **FORCE LOCK** (Sillynull)*

Forcefully **ROBLOXLOCK** an instance.

In Roblox Studio:

1. Create a character or a representation of one using a **Model** with a **Humanoid** and **HumanoidRootPart** inside it. Make sure it is able to pick up a **Hat/Accessory**.
2. Put the **Model** into a **ROBLOXLOCKED** instance.
3. Upload this setup to ROBLOX as a Module.

In Game:

1. (Recommended) Run the next steps in **Stepped** + **TWEEN PRIORITY** + **SN{80}**.
2. Use **FORCE CLONE** to create a new instance of the **Model** into **Workspace**.
3. Create a dummy **Hat**/**Accessory** in a container (**C**).
4. (Recommended) run the next steps in **C.DescendantRemoving**.
5. Parent any target instance into the Hat.
6. In **Heartbeat**, it is expected that the target instances will have already been inserted in the target **Model** and are **ROBLOXLOCKED**.

*Unfortunately, this method can only occur in the Simulation Phase of the Task Scheduler (between **Stepped TPSN80** and **Heartbeat**). However, if the antideath is not careful enough with its indexing and pcalling, it is very likely that it permanently dies.*

XIX. & MIRAGE

"such stubbornness."

"All Strengths are but a mere Delusion."

Denies the replication of the **Parent** property of an instance in the game, letting it retain across all clients while being non-existent on the server (or vice versa), akin to a replication desync. It is effectively CR-parenting done from the server.

Utilize a **Freezer** instance (either **Lighting** or **Teams**).

*The **Freezer** is not actually present as a descendant of the game on the client. Due to it being a **Service**, it is likely that the **Parent** property is not replicated across clients, and this is what causes this effect.*

We first have to define a sub-method we can call **FGC** (Force GC Client):

- **FGC** force garbage collects any instance that is in nil on the client.
- **SetAttribute** on a random persistent instance or **Service** (like **Workspace**) and then set it to nil.
- After extensive testing, using cached attribute string values (that is, strings that were used in a previous frame) and repeating it around 3000 times is an optimal way to make sure that Mirage is activated on the **Freezer** (explained below).

    *Not using repeated string values will produce an effect where it "charges up" its consistency across some frames (0-99% chance of **Mirage Activation** happening) until it reaches a point where it becomes fully consistent. From this point onward, the server lag will continue increasing until you stop for a few frames. Then, the process will restart.*

- This method can be very laggy, and it is recommended to do **MIRAGE** once every 2 frames so that the server doesn't get flooded too much from the attribute spam, which is not too bad of a sacrifice.

  *It is currently unknown why spamming attributes yields way more lag than expected. Most recent testing shows that **SignalBehavior Deferred** does not have this issue, however further research is needed.*

There are two parts to this method:

- **Mirage Activation**

  This is the state where the existence of the **Freezer** instance itself is completely erased from the client, and **Mirage Trigger** can be successfully executed on it.

  *After 1 frame, all instances that are nil on the client are eligible for garbage collection. **FGC** forces this process to happen immediately.*
- **Mirage Trigger**

  This is when the target instance's **Parent** property is desynced across clients, allowing it to be erased on the server whilst being intact on the client.

STEPS:

In Roblox Studio:

1. Insert the **Freezer** into a container.
2. Upload this setup to ROBLOX as a Module.

In Game:

1. Use **FORCE CLONE** to clone the container.

3. Parent this container somewhere that replicates to clients.
4. **Mirage Activation**: Wait for 1 frame. Alternatively, use **FGC** for instant activation (at the cost of lag).
    - ***WARNING: If you choose to wait 1 frame instead, there is a glaring weakness where the Freezer can be affected by ANTI-GC -> CLIENT.***
5. **Mirage Trigger**: Parent the target instance to **Freezer**.
6. Parent it to anywhere you want (that is replicated across clients).
7. Parent it to nil. (Additionally, you can **Destroy** the part after, however this will forever be on client until rejoin).

    Because of the method, this part actually doesn't get replicated to the client!

- ***WARNING: Because of the existence of the stupid ConnectParallel having replication priority AND being able to get instances through game.DescendantAdded even if they were already parented back to nil, two new weaknesses must be taken note of:***
    - ***Even if the instance is in nil, property changes in the same frame are still replicated to the client.***
    - ***Parenting the instance back to the game at any point of time in the future will also update every single descendant and property. This is mainly how MIRAGE could be killed.***
- ***There is fortunately a workaround to this, explained in the METHOD COMBINATIONS section below.***

*An alternative is to directly parent the **Freezer** itself to **Workspace** using **InsertService:Insert()**, parent the instance into it, then **Destroy** the **Freezer** (it retains itself on the client in **Workspace**).*
*The main advantages of this approach is that there is no need for **FGC** or **Mirage Activation**. However that means that this will forever be on client until rejoin. We are unsure if **Remove** produces the same effect. The **Freezer** will also actually show up on the clients, which may not be good for hiding the method itself.*

VARIANT 1: INFECTION

*Anything that is parented to the **Freezer** after **Mirage Activation**, and that did not previously exist in the game and on the clients, will act as a "second" freezer that produces the same effects as the original, effectively getting "infected" with the **Mirage Activation** effect.*

*After **Mirage Activation:***

1. *Create a temporary instance (e.g. **Folder**). (Let this be **I_0**).*
2. *Parent **I_0** into **Freezer**.*
3. *Create an instance you want to infect. (Let this be **I**).*
4. *Parent **I** to **I_0**.*
5. *Parent **I** to **Freezer**. **I** can now be used as a secondary freezer instance. :stupid_nigga:*
6. *(Cleanup step) Destroy **I_0**.*
7. ***Mirage Trigger:** Replace Step 5 in the original **Mirage Trigger** with parenting the target instance to **I** instead.*

*I_0 was parented to **Freezer** in Step 2, however it does not exist on any client since it was not previously somewhere that replicates. What probably happens is that in Step 4, technically **I** was inserted into the game, so the server has to replicate **I** to all clients. However, it is in **Freezer** which does not exist on the client, thus **I** remains nil on the client. In Step 5, The client attempted to parent **I** to something that doesn't exist (**Freezer**). It is currently unknown why this results in **I** having **Mirage Activation** effects, however ~~it probably has something to do with~~ it just works lolmao cope.*

VARIANT 2: REPLICATION DENIAL

Instead of removing the target instance on the server while retaining it on the client, we do the reverse.

1. *Parent the target instance into **Freezer**.*
2. ***Mirage Activation**: Wait for 1 frame. Alternatively, use **FGC** for instant activation (at the cost of lag).*
3. *Parent the instance back to its original parent.*

*Since this involves direct tampering with the parent, it is most effective with **HYPERNULL**. This is similar to Fake-Degradation, however we are unsure whether or not it is more effective.*
*Sadly, it is barely usable with **FGC** as it is inconsistent. To have a 100% success rate you must wait for 1 frame before parenting it back to the original parent, which defeats the purpose.*

thanks a lot, HyperGonest. -sol

XX. INTERNAL VOID

Void a **ROBLOXLOCKED CanQuery = false** part.

Use **WorldRoot:Raycast** or any **Shapecast** methods on an area, with **RaycastPaams.BruteForceAllSlow** set to **true**. Use **FORCE MOVE** to void the detected instance afterwards. Repeat until there are no more detected parts in the area or if a designated limit is reached.

- ***It is important to not have anything in RaycastParams.FilterDescendantsInstances as it will make the raycast not detect the ROBLOXLOCKED part.***

*Unfortunately, this cannot work on **WorldModels** inside **ROBLOXLOCKED** instances.*

METHOD COMBINATIONS

I. *PARAEXISTENCE*

Attempts to avoid detection and targeting from any attack.

Also attempts to prevent **STALL** users' attacks from adapting to the antideath's own **STALL** amount by erasing itself right before the attacker's dead/alive check.

Exist as late as possible near **Replication Send Jobs**, and erase yourself as early as possible after.

SILENCE *or* ***HYPERNULL DETECTOR -> TIMESHIFT*** *may help with running threads as early as possible.*

II. NEGENTROPY

"It's their fault now."

A very solid, nigh-impenetrable defense utilizing a combo of **MIRAGE** and **ROBLOXLOCKED**.

Combine **ROBLOXLOCKED** instances and use **MIRAGE** on the container to disallow tampering on the antideath.

If the container instance's parent is tampered with, then call **Destroy** on the next container instance in the next frame. This will make the instances be permanently on screen until rejoin, and prevent anything from ever tampering with the instances at that particular frame.

If you don't want parts to stay on screen forever, **FORCE PARENT** can be used to parent a **BasePart/Tool/Accessory** outside the **ROBLOXLOCKED** instance and into the game, updating the state of all descendants of the container and "killing" it. The container itself will still stay on the client until rejoin, however.

*As previously mentioned, Instances that were **MIRAGE**'d and are in nil are still prone to property changes in the same frame, as these changes will still replicate. Locking the instances prevents most of this from happening.*

*III. TRINITY: **Revived** (Full Anti-read)*

⃠ 鍵は。。。 ⃠

門は開かれた。

Parent a container with **ROBLOXLOCKED** instances directly in **Workspace** on client using **MIRAGE**.

Combine **NEGENTROPY** (using **MIRAGE -> INFECTION**) and **ROBLOXLOCKED -> PERM LOCK -> WHOLE** to almost completely prevent any vulnerabilities from taking place, ***ever.***

*The way to reparent the **WHOLE** variant anywhere is to use dummy/temporary **Models** with **Humanoids**. To do **Mirage Activation** and **Mirage Trigger**, use **INFECTION** for one of the **Models**.*

*This method combination is most likely the most untamperable an antideath could be as of yet (3/12/2024). There are currently only two known ways to "break" it. It is mainly composed of two shields. We can treat **MIRAGE** as the **BLUE** shield and **PERMLOCK** as the **RED** shield.*

*To break the **RED** shield, use **ANTI-GC -> SERVER** to abuse **WHOLE**'s weakness. This is so we can actually start to interact with the antideath in the first place.*

*To "break" the **BLUE** shield, parent the instance back to the game then **Destroy** it, then stall for time with **while true do end** lmao, because if the defender does everything correctly in **NEGENTROPY** then they will detect that you have killed them and **Destroy** themselves in the next frame to be permanently on screen until rejoin.*

*There is a potential third way to break it by not allowing the Spell™ to occur using **Script Timeout**, however further testing is needed.*

God, unsealed once again, and returns even stronger than before.

IV. MIRAGE EDGE

"Slay all!"

Utilize **TRINITY: Revived** in combination with **PRIORITY V8 (?) -> CLEAVE** to make the insertion of the **NaN** part in the target model nigh undetectable.

*It is recommended to use **AddAccessory** for the container of the attacker part. **ReplaceBodyPartR15/EquipTool** both remove and override whatever body part/**Tool** is currently in the model, which may cause unintentional removal of the antideath's instances and is prone to getting detected.*

The insertion can still be detected at the point where the corrupted parts are parented out of the ROBLOXLOCKED -> PERM LOCK-ed instance (for when AssemblyLinearVelocity client movement has to occur through MIRAGE). More specifically, it can be detected by HN DETECTOR through DescendantRemoving. (This is hella obscure tho)

TRINITY BRANCHES

Around 2022, we decided to categorize methods into three categories: **SPACE**, **TIME**, and **REALITY**.

We felt that as more methods were discovered, categories were able to be formed to help decide what branch to focus on. The methods in these branches are different from attack methods (e.g. Destroy, Void, Mesh, Fake Degradation). They either only pertain to things that assist these methods, or do something else entirely different. Branches don't include methods that don't directly have an effect on defense/attack (e.g. **SILENCE**). Below are the descriptions of each branch, as well as some methods classified under them. It may not be the most robust way to categorize methods, but it's pretty cool and epic so why not XD

SPACE

- Main:
  - WorldModel
  - CanQuery false
  - Mesh Offset
- Assist:
  - Name/Class Randomization (anti-banish)
  - Property Randomization (anti-decimate)
  - ParaExistence

Methods that assist in anti-detection from an attack. Mostly used by older scripts in an era where attacks had more specific targets.

TIME

- Loops
  - Tween Priority
  - Priority V3
  - Stall
  - Anti-Sync
- Events
  - Priority Grab
  - Hypernull
  - Supernull
  - Hypernull Detector

Methods that assist in the “speed” of a defense/attack. Speed is a common misconception among attacks. Time branch methods pertain more to the concept of gaining replication priority in the Task Scheduler.

This branch is probably the most popular of the three, as it is both reliable and accessible, and helps with various situations, acting as a “second line of defense” after **SPACE**’s anti detection mechanics.

REALITY

- Main:
    - ROBLOXLOCKED
    - Priority V8 (?)
    - Mirage
- Assist:
    - Force Move
    - Force Parent
    - Internal Eliminate
    - Internal Void

Methods that produce effects very close to the core of the ROBLOX engine. It is perhaps the most obscure branch of the three. Methods under this branch provide the most power to a defense/attack, as they are something akin to a gaslight or joke method – that is, things that seem fake, ***but are actually real.***

We serve nothing but the truth

Truth and only truth

amidst a sea of lies.

Heaven's Cathedral

signing out