Repainting

Introduction

We define repainting as: **script behavior causing historical vs realtime calculations or plots to behave differently**.

Repainting behavior is widespread and can be caused by many factors. Following our definition, our estimate is that more than 95% of indicators in existence repaint. Widely used indicators like MACD and RSI, for example, exhibit one form of repainting because they show one value on historical bars, yet when running in realtime they will produce results that constantly fluctuate until the realtime bar closes. They thus behave differently on historical and realtime bars. This does not necessarily make them less useful in all contexts, nor prevent knowledgeable traders from using them. Who would think of discrediting a volume profile indicator, for example because it updates in real time, and so repaints?

The different types of repainting we discuss in this page can be divided this way:

- **Widespread and often acceptable**: recalculation during the realtime bar (most classic indicators like MACD, RSI, and the vast majority of indicators in the Community Scripts, scripts using repainting request.security() calls, etc.). There is often nothing wrong in using such scripts, provided you understand how they work. If you elect to use these scripts to issue alerts or trade orders, however, then you should know if they are being generated using the realtime or confirmed values, and decide for yourself if the script's behavior meets your requirements.
- **Misleading**: plotting in the past, calculating results in realtime that cannot be replicated on historical bars, relocating past events (Ichimoku, most pivot scripts, most strategies using `calc_on_evey_tick = true`,

scripts using repainting request.security() calls when their values are plotted on historical bars, as their behavior will not be the same in realtime, most scripts using varip, most scripts using timenow, some scripts using `barstate.*` variables).
- **Unacceptable**: scripts using future information, strategies running on non-standard charts, scripts using realtime intrabar timeframes to generate alerts or orders.
- **Unavoidable**: revision of historical feeds by data suppliers, varying starting bar on historical bars.

The first two types of repainting can be perfectly acceptable if:

1. You are aware of the behavior.
2. You can live with it, or
3. You can circumvent it.

It should now be clear to you that not **all** repainting behavior is inherently wrong and should be avoided at all cost. In many situations, repainting indicators are exactly what's needed. What's important is to know when repainting behavior is **not** acceptable to you. To avoid such situations, you must understand exactly how the tools you use work, or how you should design the ones you build. If you publish scripts, any potentially misleading repainting behavior should be mentioned along with the other limitations of your script.

> **Note**
> We will not discuss the perils of using strategies on non-standard charts, as this problem is not related to repainting. See the Backtesting on Non-Standard Charts: Caution! script for a discussion of the subject.

For script users

You can very well decide to use repainting indicators if you understand how they behave, and that behavior meets your trading methodology's requirements. Don't be one of those newcomers to trading who slap "repaint" sentences on published scripts as if it discredits them. Doing so only reveals your incomprehension of the subject.

The question "Does it repaint?" means nothing. Consequently, it cannot be answered in a meaningful way. Why? Because it needs to be qualified. Instead, one could ask:

- Do you wait for the realtime bar to close before displaying your entry/exit markers?
- Do alerts wait for the end of the realtime bar before triggering?
- Do the higher timeframe plots repaint (which means they won't plot the same way on realtime bars as they do on historical bars)?
- Does your script plot in the past (as most pivot or zigzag scripts will do)?
- Does your strategy use `calc_on_every_tick = true`?
- Will your indicator display in realtime the same way it does on historical bars?
- Are you fetching future information with your request.security() calls?

What's important is that you understand how the tools you use work, and if their behavior is compatible with your objectives, repainting or not. As you will learn if you read this page, repainting is a complex matter. It has many faces and many causes. Even if you don't program in Pine Script™, this page will help you understand the array of causes that can lead to repainting, and hopefully enable more meaningful discussions with script authors.

For Pine Script™ programmers

As we discussed in the previous section, not all types of repainting behavior need to be avoided at all costs, and as we will see in the following text, some can't. We hope this page helps you better understand the dynamics at play, so that you can make better design decisions concerning your trading tools. This page's content should help you avoid making the most common coding mistakes that lead to repainting or misleading plots.

Whatever your design decisions are, if you publish your script, you should explain them to traders so they can understand how your script behaves.

We will survey three broad categories of repainting causes:

- Historical vs realtime calculations
- Plotting in the past
- Dataset variations

Historical vs realtime calculations

Fluid data values

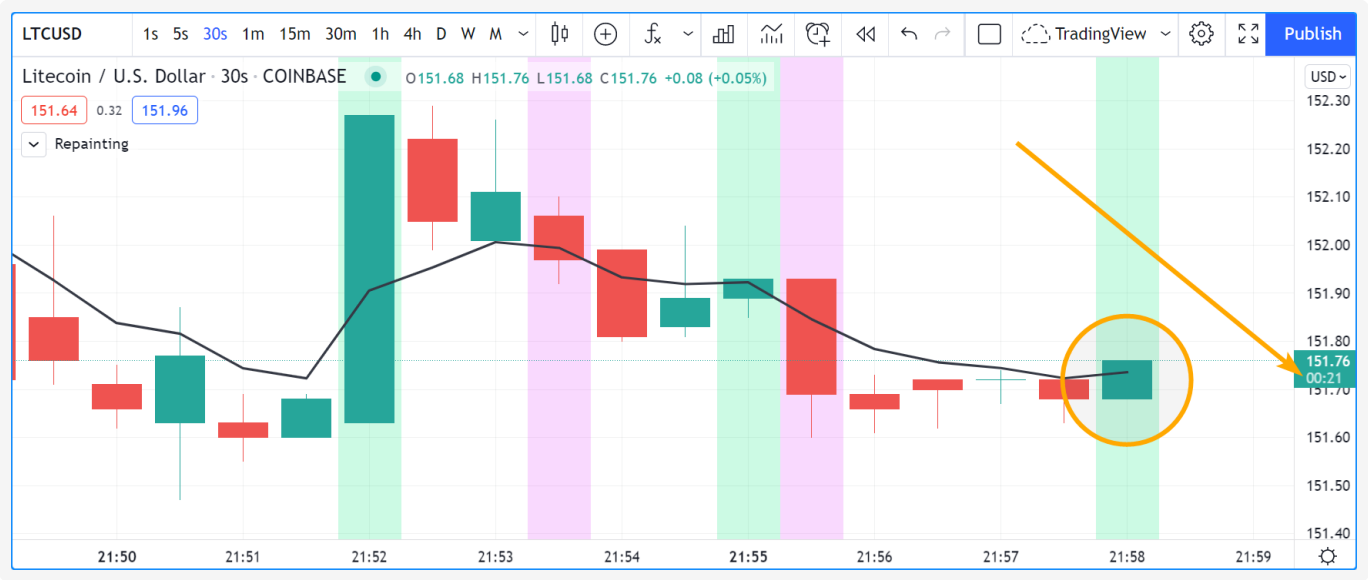
Historical data does not include records of intermediary price movements on bars; only open, high, low and close values (OHLC).

On realtime bars (bars running when the instrument's market is open), however, the high, low and close values are not fixed; they can change values many times before the realtime bar closes and its HLC values are fixed. They are *fluid*. This leads to a script sometimes working differently on historical data and in real time, where only the open price will not change during the bar.

Any script using values like high, low and close in realtime is subject to producing calculations that may not be repeatable on historical bars — thus repaint.

Let's look at this simple script. It detects crosses of the close value (in the realtime bar, this corresponds to the current price of the instrument) over and under an EMA:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma)
xDn = ta.crossunder(close, ma)
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```



Note that:

- The script uses bgcolor() to color the background green when close crosses over the EMA, and red on crosses under the EMA.
- The screen snapshot shows the script in realtime on a 30sec chart. A cross over the EMA has been detected, thus the background of the realtime bar is green.
- The problem here is that nothing guarantees this condition will hold true until the end of the realtime bar. The

arrow points to the timer showing that 21 seconds remain in the realtime bar, and anything could happen until then.
- We are witnessing a repainting script.

To prevent this repainting, we must rewrite our script so that it does not use values that fluctuate during the realtime bar. This will require using values from a bar that has elapsed (typically the preceding bar), or the open price, which does not vary in realtime.

We can achieve this in many ways. This method adds a `and barstate.isconfirmed` condition to our cross detections, which requires the script to be executing on the bar's last iteration, when it closes and prices are confirmed. It is a simple way to avoid repainting:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma) and barstate.isconfirmed
xDn = ta.crossunder(close, ma) and barstate.isconfirmed
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

This uses the crosses detected on the previous bar:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(close, ma)[1]
xDn = ta.crossunder(close, ma)[1]
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

This uses only confirmed close and EMA values for its calculations:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close[1], 5)
xUp = ta.crossover(close[1], ma)
xDn = ta.crossunder(close[1], ma)
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

This detects crosses between the realtime bar's open and the value of the EMA from the previous bars. Notice that the EMA is calculated using close, so it repaints. We must ensure we use a confirmed value to detect crosses, thus `ma[1]` in the cross detection logic:

```
//@version=5
indicator("Repainting", "", true)
ma = ta.ema(close, 5)
xUp = ta.crossover(open, ma[1])
xDn = ta.crossunder(open, ma[1])
plot(ma, "MA", color.black, 2)
bgcolor(xUp ? color.new(color.lime, 80) : xDn ? color.new(color.fuchsia, 80) : na)
```

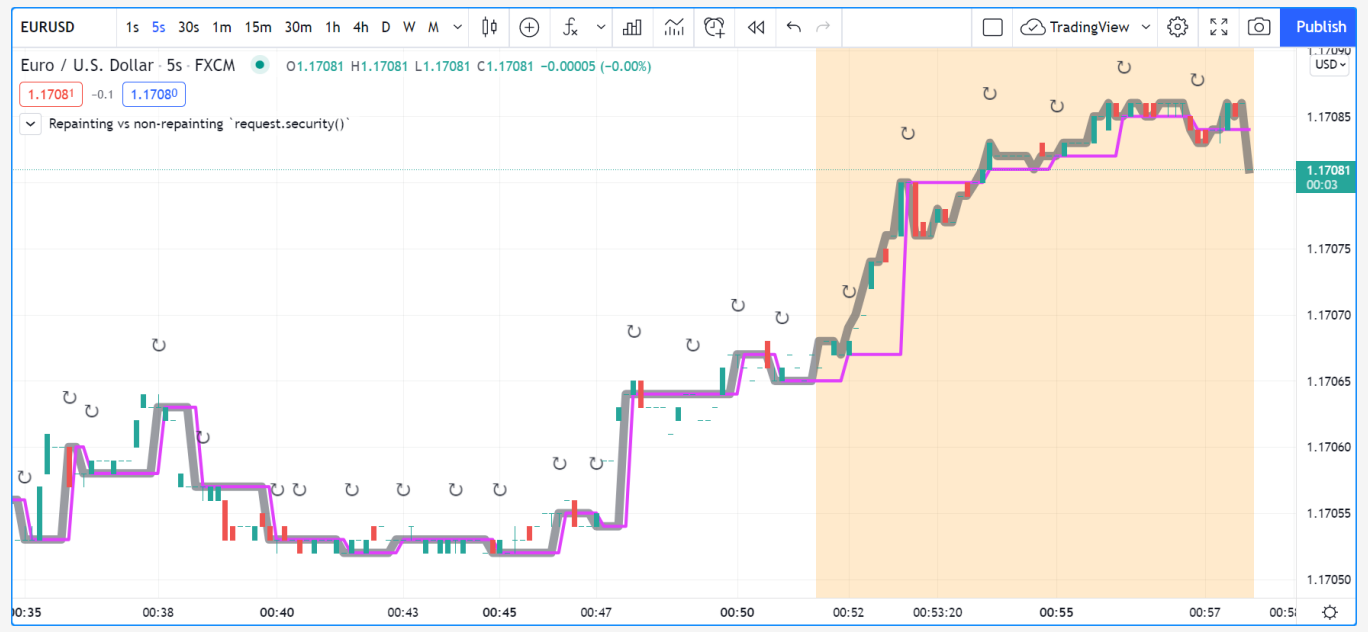
Note that all these methods have one thing in common: while they prevent repainting, they will also trigger signals later than repainting scripts. This is an inevitable compromise if one wants to avoid repainting. You just can't have your cake and eat it too.

Repainting `request.security()` calls

The data fetched with request.security() will differ on historical and realtime bars if the function is not used in the correct manner. Repainting request.security() calls will produce historical data and plots that cannot be replicated in realtime. Let's look at a script showing the difference between repainting and non-repainting request.security() calls:

```
//@version=5
indicator("Repainting vs non-repainting `request.security()`", "", true)
var BLACK_MEDIUM = color.new(color.black, 50)
var ORANGE_LIGHT = color.new(color.orange, 80)

tfInput = input.timeframe("1")

repaintingClose = request.security(syminfo.tickerid, tfInput, close)
plot(repaintingClose, "Repainting close", BLACK_MEDIUM, 8)

indexHighTF = barstate.isrealtime ? 1 : 0
indexCurrTF = barstate.isrealtime ? 0 : 1
nonRepaintingClose = request.security(syminfo.tickerid, tfInput, close[indexHighTF])[ir
plot(nonRepaintingClose, "Non-repainting close", color.fuchsia, 3)

if ta.change(time(tfInput))
    label.new(bar_index, na, "↻", yloc = yloc.abovebar, style = label.style_none, textc
bgcolor(barstate.isrealtime ? ORANGE_LIGHT : na)
```

This is what its output looks like on a 5sec chart that has been running the script for a few minutes:



Note that:

- The orange background identifies the realtime bar, and elapsed realtime bars.
- A black curved arrow indicates when a new higher timeframe comes in.
- The thick gray line shows the repainting request.security() call used to initialize `repaintingClose`.
- The fuchsia line shows the non-repainting request.security() call used to initialize `nonRepaintingClose`.
- The behavior of the repainting line is completely different on historical bars and in realtime. On historical bars, it shows the new value of a completed timeframe on the close of the bar where it completes. It then stays stable until another timeframe completes. The problem is that in realtime, it follows the **current** close price, so it moves all the time and changes on each bar.
- The behavior of the non-repainting fuchsia line, in contrast, behaves exactly the same way on historical bars and in realtime. It updates on the bar following the completion of the higher timeframe, and doesn't move until the

bar after another higher timeframe completes. It is more reliable and does not mislead script users. Note that while new higher timeframe data comes in at the close of historical bars, it will be available on the open of the same bar in realtime.

This script shows a `nonRepaintingSecurity()` function that can be used to do the same as our non-repainting code in the previous example:

```
//@version=5
indicator("Non-repainting `nonRepaintingSecurity()`", "", true)

tfInput = input.timeframe("1")

nonRepaintingSecurity(sym, tf, src) =>
    request.security(sym, tf, src[barstate.isrealtime ? 1 : 0])[barstate.isrealtime ? (

nonRepaintingClose = nonRepaintingSecurity(syminfo.tickerid, "1", close)
plot(nonRepaintingClose, "Non-repainting close", color.fuchsia, 3)
```

Another way to produce non-repainting higher timeframe data is this, which uses an offset of `[1]` on the series, and `lookahead`:

```
nonRepaintingSecurityAlternate(sym, tf, src) =>
    request.security(sym, tf, src[1], lookahead = barmerge.lookahead_on)
```

It will produce the same non-repainting behavior as `nonRepaintingSecurity()`. Note that the `[1]` offset to the series and the use of `lookahead = barmerge.lookahead_on` are interdependent. One **cannot** be removed without compromising the functionality of the function. Also note that occasional one-bar variations between when the `nonRepaintingSecurity()` and `nonRepaintingSecurityAlternate()` values come in on historical bars are to be expected.

Using `request.security()` at lower timeframes

Some scripts use request.security() to request data from a timeframe **lower** than the chart's timeframe. This can be useful when functions specifically designed to handle intrabars at lower timeframes are sent down the timeframe. When this type of user-defined function requires the detection of the intrabars' first bar, as most do, the technique will only work on historical bars. This is due to the fact that realtime intrabars are not yet sorted. The impact of this is that such scripts cannot reproduce in real time their behavior on historical bars. Any logic generating alerts, for example, will be flawed, and constant refreshing will be required to recalculate elapsed realtime bars as historical bars.

When used at lower timeframes than the chart's without specialized functions able to distinguish between intrabars, request.security() will only return the value of the **last** intrabar in the dilation of the chart's bar, which is usually not useful, and will also not reproduce in real time, so lead to repainting.

For all these reasons, unless you understand the subtleties of using request.security() at lower timeframes than the chart's, it is best to avoid using the function at those timeframes. Higher-quality scripts will have logic to detect such anomalies and prevent the display of results which would be invalid when a lower timeframe is used.

Future leak with `request.security()`

When request.security() is used with `lookahead = barmerge.lookahead_on` to fetch prices without offsetting the series by `[1]`, it will return data from the future on historical bars, which is dangerously misleading.

While historical bars will magically display future prices before they should be known, no lookahead is possible in realtime because the future there is unknown, as it should, so no future bars exist.

This is an example:

```
// FUTURE LEAK! DO NOT USE!
//@version=5
indicator("Future leak", "", true)
futureHigh = request.security(syminfo.tickerid, "D", high, lookahead = barmerge.lookahe
plot(futureHigh)
```

Note how the higher timeframe line is showing the timeframe's high value before it occurs. The solution is to use the function like we do in our `nonRepaintingSecurity()` shown earlier.

Public scripts using this misleading technique will be moderated.

`varip`

Scripts using the varip declaration mode for variables (see our section on varip for more information) save information across realtime updates, which cannot be reproduced on historical bars where only OHLC information is available. Such scripts may be useful in realtime, including to generate alerts, but their logic cannot be backtested, nor can their plots on historical bars reflect calculations that will be done in realtime.

Bar state built-ins

Scripts using bar states may or may not repaint. As we have seen in the previous section, using barstate.isconfirmed is actually one way to **avoid** repainting that **will** reproduce on historical bars, which are always "confirmed". Uses of other bar states such as barstate.isnew, however, will lead to repainting. The reason is that on historical bars, barstate.isnew is `true` on the bar's close, yet in realtime, it is `true` on the bar's open. Using the other bar state variables will usually cause some type of behavioral discrepancy between historical and realtime bars.

`timenow`

The timenow built-in returns the current time. Scripts using this variable cannot show consistent historical and realtime behavior, so they necessarily repaint.

Strategies

Strategies using `calc_on_every_tick = true` execute on each realtime update, while strategies run on the close of historical bars. They will most probably not generate the same order executions, and so repaint. Note that when this happens, it also invalidates backtesting results, as they are not representative of the strategy's behavior in realtime.

Plotting in the past

Scripts detecting pivots after 5 bars have elapsed will often go back in the past to plot pivot levels or values on the actual pivot, 5 bars in the past. This will often cause unsuspecting traders looking at plots on historical bars to infer that when the pivot happens in realtime, the same plots will apppear on the pivot when it occurs, as opposed to when it is detected.

Let's look at a script showing the price of high pivots by placing the price in the past, 5 bars after the pivot was detected:

```
//@version=5
indicator("Plotting in the past", "", true)
pHi = ta.pivothigh(5, 5)
if not na(pHi)
    label.new(bar_index[5], na, str.tostring(pHi, format.mintick) + "\n🅰🅰", yloc = yl
```

Note that:

- This script repaints because an elapsed realtime bar showing no price may get a price placed on it if it is identified as a pivot, 5 bars after the actual pivot occurs.
- The display looks great, but it can be misleading.

The best solution to this problem when developing script for others is to plot **without** an offset by default, but give the option for script users to turn on plotting in the past through inputs, so they are necessarily aware of what the script is doing, e.g.:

```
//@version=5
indicator("Plotting in the past", "", true)
plotInThePast = input(false, "Plot in the past")
pHi = ta.pivothigh(5, 5)
if not na(pHi)
    label.new(bar_index[plotInThePast ? 5 : 0], na, str.tostring(pHi, format.mintick) +
```

Dataset variations

Starting points

Scripts begin executing on the chart's first historical bar, and then execute on each bar sequentially, as is explained in this manual's page on Pine Script™'s execution model. If the first bar changes, then the script will often not calculate the same way it did when the dataset began at a different point in time.

The following factors have an impact on the quantity of bars you see on your charts, and their *starting point*:

- The type of account you hold
- The historical data available from the data supplier
- The alignment requirements of the dataset, which determine its *starting point*

These are the account-specific bar limits:

- 20000 historical bars for the Premium plan.
- 10000 historical bars for Pro and Pro+ plans.
- 5000 historical bars for other plans.

Starting points are determined using the following rules, which depend on the chart's timeframe:

- **1 - 14 minutes**: aligns to the beginning of a week.
- **15 - 29 minutes**: aligns to the beginning of a month.
- **30 minutes and higher**: aligns to the beginning of a year.

As time goes by, these factors cause your chart's history to start at different points in time. This often has an impact on your scripts calculations, because changes in calculation results in early bars can ripple through all the other bars in the dataset. Using functions like ta.valuewhen(), ta.barssince() or ta.ema(), for example, will yield results that vary with early history.

Revision of historical data

Historical and realtime bars are built using two different data feeds supplied by exchanges/brokers: historical data, and realtime data. When realtime bars elapse, exchanges/brokers sometimes make what are usually small adjustments to bar prices, which are then written to their historical data. When the chart is refreshed or the script is re-executed on those elapsed realtime bars, they will then be built and calculated using the historical data, which will contain those usually small price revisions, if any have been made.

Historical data may also be revised for other reasons, e.g., for stock splits.

Plots

Sessions